UCRL-ID-151712

Scientific Component Technology Initiative

Scott Kohn, Bill Bosl, Tammy Dahlgren, Tom Epperly, Gary Kumfert, and Steve Smith

February 7, 2003

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

LDRD Final Report for 00-SI-002

Scientific Component Technology Initiative

Principal Investigator: Scott Kohn
Center for Applied Scientific Computing
Computations Directorate

Co-Investigators: Bill Bosl, Tammy Dahlgren**,** Tom Epperly, Gary Kumfert, and Steve Smith
Center for Applied Scientific Computing
Computations Directorate

Funding Summary: FY00: $450K
FY01: $800K
FY02: $625K

Project Overview

The laboratory has invested a significant amount of resources towards the development of high-performance scientific simulation software, including numerical libraries, visualization, steering, software frameworks, and physics packages. Unfortunately, because this software was not designed for interoperability and re-use, it is often difficult to share these sophisticated software packages among applications due to differences in implementation language, programming style, or calling interfaces.

This LDRD Strategic Initiative investigated and developed software component technology for high-performance parallel scientific computing to address problems of complexity, re-use, and interoperability for laboratory software. Component technology is an extension of scripting and object-oriented software development techniques that specifically focuses on the needs of software interoperability. Component approaches based on CORBA, COM, and Java technologies are widely used in industry; however, they do not support massively parallel applications in science and engineering. Our research focused on the unique requirements of scientific computing on ASCI-class machines, such as fast in-process connections among components, language interoperability for scientific languages, and data distribution support for massively parallel SPMD components.

Activities and Technical Achievements

Over the course of this project, we have investigated, developed, and demonstrated scientific component technology in the following areas.

Babel Language Interoperability Technology

We have developed a tool called **Babel** that addresses language interoperability issues for high-performance parallel scientific software. Its purpose is to enable the creation, description, and distribution of language independent software libraries. **Babel** uses Interface Definition Language (IDL) techniques. An IDL describes the calling interface (but not the implementation) of a particular software library. IDL tools such as **Babel** use this interface description to generate glue code that allows a software library implemented in one supported language to be called from any other supported language. We have designed a Scientific Interface Definition Language (SIDL) that addresses the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. **Babel** currently supports Fortran 77, C, C++, Python, client-side Java, and some Fortran 90.

Web-Based Component Repository

We have developed two web-based tools to simplify the sharing of component software and the development of community software interface standards. **Alexandria** is a component repository for storing component software and SIDL interface descriptions and is being deployed as the repository in the CCA (Common Component Architecture) community component infrastructure. **Quorum** is a web-based voting server that is in use by the CCA working group for establishing software interface standards. We have deployed both **Alexandria** and **Quorum** on the LLNL externally visible green network at http://www-casc.llnl.gov.

Collaboration with *hypre* Scalable Linear Solvers Project

In collaboration with members of the *hypre* development team, we have integrated some of the **Babel** language interoperability technology into *hypre*. The *hypre* library is a suite of parallel scalable linear solvers and preconditioners. It supplies critical solver technology to the ASCI program and is used by all three main ASCI codes at LLNL and by collaborators at other national laboratories. In the long term, the *hypre* team plans to migrate to a software architecture that uses **Babel** as an integral part of the library. In this design, **Babel** will provide the primary interface to *hypre* for all languages supported by the library. This approach provides the maximum benefit to the *hypre* library.

Demonstration Project in Laser Plasma Physics

In collaboration with the ALPS (Adaptive Laser Plasma Simulator) team at LLNL, we demonstrated the use of Babel in a complex parallel simulation code. ALPS is an adaptive mesh refinement simulation code that investigates the interaction of a laser with plasma for inertial confinement fusion. Our modified ALPS code uses Python as a scripting language and mixes C++, Fortran, and Python. For example, from the Python scripting layer, we can call the application framework written in C++, which in turn calls a numerical routine written in Fortran, which in turn calls a laser boundary condition module in Python. This interoperability allows a scientist to rapidly prototype new boundary condition modules in Python without recompiling or linking.

DOE Common Component Architecture SciDAC

We are collaborating on community technology standards with members of the DOE's Common Component Architecture (CCA) working group (see http://www.cca-forum.org/). The DOE Office of Science has selected the CCA as one of the recipients of a SciDAC (Scientific Discovery through Advanced Computing) award, a five-year $3.5M/yr research effort consisting of DOE laboratories and academic partners intended to deliver component technology to computational simulation efforts within the DOE. **Babel** plays a central role in the CCA SciDAC center. The CCA uses the language interoperability technology developed at LLNL as a foundation for the community common component infrastructure. **Babel** will play a critical role linking SciDAC numerical and meshing libraries, typically written in C or C++, with SciDAC applications written in Fortran 90.

Research Papers and Reports

The following four research papers and reports summarize the scope of our activities under this strategic initiative. These papers are attached to this final report.

The first paper provides an overview of the approach and technology used to develop the Babel and Alexandria tools. This paper was an invited presentation at the 2000 *Working Conference on Software Architectures for Scientific Computing Applications* sponsored by the International Federation for Information Processing in Ottawa, Canada. Conference information is available at http://www.nsc.liu.se/~boein/ifip/woco8.html. The release number is UCRL-JC-140549.

> Tom Epperly, Scott Kohn, and Gary Kumfert. **Component Technology for High-Performance Scientific Simulation Software.** Working Conference on "Software Architectures for Scientific Computing Applications", International Federation for Information Processing, Ottawa, Ontario, Canada, October 2-4, 2000.

Abstract: We are developing scientific software component technology to manage the complexity of modern, parallel simulation software and increase the interoperability and re-use of scientific software packages. In this paper, we describe a language interoperability tool named **Babel** that enables the creation and distribution of language-independent software libraries using interface definition language (IDL) techniques. We have created a scientific IDL that focuses on the unique interface description needs of scientific software, such as complex numbers, dense multidimensional arrays, and parallel distributed objects. Preliminary results indicate that in addition to language interoperability, this approach provides useful tools for the design of modern object-oriented scientific software libraries. We also describe a web-based component repository called **Alexandria** that facilitates the distribution, documentation, and re-use of scientific components and libraries.

The second paper describes the use of our component technology tools in the *hypre* scalable linear solvers library. This paper was presented at the *10th SIAM Conference on Parallel Processing*. The conference web site is http://www.siam.org/meetings/pp01. The release number is UCRL-JC-104349.

Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. **Divorcing Language Dependencies from a Scientific Software Library.** 10th SIAM Conference on Parallel Processing, Portsmouth, VA, March 12-14, 2001.

Abstract: This paper describes the ideas, process, and results of the first year in an ongoing collaboration between members of the Components Project and the *hypre* Project in the Center for Applied Scientific Computing (CASC) in Lawrence Livermore National Laboratory. The Components Project has developed a tool called **Babel** that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation and distribution of language independent software libraries. *Hypre* is a parallel, scalable scientific library of linear solvers and preconditioners. By using **Babel** tools on *hypre* in this collaboration, we found that Babel enables better software design and is an effective tool for producing language independent scientific software libraries at a negligible performance overhead.

The third paper studies the performance issues associated with component technology for high-performance scientific computing. This paper was published in collaboration with our SciDAC co-investigators in the Common Component Architecture working group. This paper was presented at the 2002 *Workshop on Performance Optimimization via High-Level Languages and Libraries*. Conference information is available at the web site http://www.ece.lsu.edu/jxr/ics02workshop.html. The release number is UCRL-JC-148723.

David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly, **A Component Architecture for High-Performance Computing**, in Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02), New York, NY. June 22 ,2002.

Abstract: The Common Component Architecture (CCA) provides a means for developers to manage the complexity of large-scale scientific software systems and to move toward a "plug and play" environment for high-performance computing. The CCA model allows for a direct connection between components within the same process to maintain performance on inter-component calls. It is neutral with respect to parallelism, allowing components to use whatever means they desire to communicate within their parallel "cohort." We will discuss in detail the importance

of performance in the design of the CCA and will analyze the performance costs associated with features of the CCA.

The final report discusses the use of the Babel language interoperability technology in the context of a LLNL laser plasma application. This is an unpublished technical report. The release number is UCRL-JC-150544.

William J. Bosl, Steven G. Smith, Tamara Dahlgren, Thomas Epperly, Scott Kohn, and Gary Kumfert. **Component Technology for Laser Plasma Simulation.** September 23 , 2002.

Abstract: This paper will discuss the application of high performance component software technology developed for a complex physics simulation development effort. The primary tool used to build software components is called Babel and is used to create language-independent libraries for high performance computers. Components were constructed from legacy code and wrapped with a thin Python layer to enable run-time scripting. Low-level components in Fortran, C++, and Python were composed directly as Babel components and invoked interactively from a parallel Python script.

Preprint
UCRL-JC-140549

Component Technology for High-Performance Scientific Simulation Software

T. Epperly, S. Kohn, and G. Kumfert

This article was submitted to the
Working Conference on "Software Architectures for Scientific Computing Applications, International Federation of Information Processing, Ottawa, Ontario, Canada, October 2-4, 2000.

U.S. Department of Energy
Lawrence Livermore National Laboratory

October 2, 2000

COMPONENT TECHNOLOGY FOR HIGH-PERFORMANCE SCIENTIFIC SIMULATION SOFTWARE*

Tom Epperly, Scott Kohn, and Gary Kumfert
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
tepperly@llnl.gov
skohn@llnl.gov
kumfert@llnl.gov

Abstract We are developing scientific software component technology to manage the complexity of modern, parallel simulation software and increase the interoperability and re-use of scientific software packages. In this paper, we describe a language interoperability tool named **Babel** that enables the creation and distribution of language-independent software libraries using interface definition language (IDL) techniques. We have created a scientific IDL that focuses on the unique interface description needs of scientific software, such as complex numbers, dense multidimensional arrays, and parallel distributed objects. Preliminary results indicate that in addition to language interoperability, this approach provides useful tools for the design of modern object-oriented scientific software libraries. We also describe a web-based component repository called **Alexandria** that facilitates the distribution, documentation, and re-use of scientific components and libraries.

Keywords: component technology, language interoperability, software repository, parallel high-performance scientific software

*Work performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. Work funded by LLNL LDRD grant 00-SI-002 and the ACTS program of the DOE Office of Science.

1. MOTIVATION

Numerical simulations play a vital role as a basic research tool for understanding fundamental physical processes. As simulations become increasingly sophisticated and complex, no single person—or even single institution—can develop scientific software in isolation. Development teams rarely possess sufficient resources and scientific expertise in all required domains to successfully create a complex application from scratch. Instead, physicists, chemists, mathematicians, and computer scientists concentrate on developing software in their domain of expertise. Computational scientists create simulations by combining these individual software pieces.

In collaboration with the Common Component Architecture forum [1], we are developing software component technology for high-performance parallel scientific computing. The goal of this effort is to improve the software development processes of scientific codes by using proven techniques and technology from industry. Component technology addresses technological barriers to software re-use and integration, such as incompatibilities in programming languages, interface descriptions, and physical deployment. By removing such barriers, component approaches will allow computational scientists to concentrate on building more sophisticated numerical simulations and reduce effort wasted integrating incompatible software.

In this paper, we describe our recent work in two areas of component technology: language interoperability and a component repository. As part of our language interoperability efforts, we are developing a tool called **Babel** to enable the creation and distribution of language independent software libraries. To use **Babel**, library developers describe their software interfaces in a Scientific Interface Definition Language (SIDL). **Babel** uses this SIDL interface description to automatically generate "glue code" that enables the software library to be called from any supported language. We have also designed and implemented a prototype web-based repository called **Alexandria** to encourage the distribution and reuse of scientific computing software components and libraries. **Alexandria** provides a convenient web-based delivery system and thus lowers the barrier to adopting component technology.

This paper is organized as follows. Section 2 surveys component technology approaches for scientific computing and discusses related work. Section 3 discusses our language interoperability approach, modifications necessary for the scientific domain, the **Babel** tool, and experiences using **Babel** in a high-performance scientific software library. Section 4 introduces the **Alexandria** web-based component repository and its implementation architecture. Finally, Section 5 summarizes the contributions of this work and discusses future research directions for the scientific component community.

2. SCIENTIFIC COMPONENT TECHNOLOGY

Component technology [25] is an extension of object-oriented software technology that focuses on the issues of software interoperability and re-use. Component technology provides language independence, compiler independence, and seamless access to distributed object resources. Component technology is more than object-oriented approaches, software modules, scripting [3, 4], or software frameworks [7, 8, 10, 14]; however, component approaches do make use of these other related technologies. A software framework may be created within a component architecture to address a particular application domain. Scripting languages may be used as an integration language to connect existing software components.

Industry has created component technology to address issues of interoperability due to different programming languages, the complexity of applications developed using third-party software, and the incremental evolution of large legacy software. There are three common component technology standards in the business community: COM [12], JavaBeans [24], and CORBA [19]. COM is Microsoft's component standard that forms the basis for interoperability among all Windows-based applications. Microsoft recently introduced a new component initiative called .NET [18] that combines ideas from COM and `Java` and will likely be the future of Microsoft technology. Sun Microsystems has developed JavaBeans and Enterprise JavaBeans [23] based on the `Java` programming language. CORBA, by the Object Management Group (OMG), is a cross-platform distributed object specification that supports the interaction of complex objects written in different programming languages distributed across a network of computers.

Component technologies such as CORBA, COM, and JavaBeans have been very successful in industry; unfortunately, they are designed for the business environment and do not address many of the issues associated with large-scale parallel scientific computing. For example, industry approaches do not address data distribution support for massively parallel SPMD components.

We believe that a successful component technology for scientific simulation must address four issues: language interoperability, common component behavior, physical deployment standards, and support for distributed parallel communication. The work presented in this paper addresses only a small part of the overall component technology solution. Community collaborative work such as that by the Common Component Architecture (CCA) [1] forum and others is essential. In the following, we review related component technology work in the scientific community.

Both CORBA [19] and COM [12] address language interoperability through the use of an Interface Definition Language (IDL). An IDL describes the interface of a software component using a new descriptive language that is indepen-

dent of any particular programming language. We follow a similar approach in our language interoperability work, which is presented in Section 3. IDL technology has the advantage that, in some sense, all languages are equal, and any language may call any other language. The primary disadvantage of an IDL approach is that the developer must write a separate interface description of the software library and then must follow certain programming conventions that map the interface description into the programming language. Automatic wrapping approaches such as SWIG [3] or SILOON [17] support language interoperability without requiring a separate IDL description but are typically limited to the case of a scripting language (such as `Python`) calling a compiled language (such as `C` or `C++`). In contrast, IDL approaches allow method invocations in both directions.

Beyond language interoperability, component architectures typically require that all components support some common set of behaviors. Common behaviors are important for the discovery of component capabilities (e.g., *"What interfaces do you export?"*) required by GUI development tools and problem solving environments [6, 13, 20]. For example, the CCA specification requires that all CCA components support the notion of a *port* [1]. Ports describe the interfaces used by and provided by a component. Our IDL technology plays a role as a mechanism for describing component port interfaces.

Component problem solving environments (PSEs) may also require standards for describing the physical deployment of component software. For example, CCAT [6] employs an XML [28] component deployment descriptor that enables the PSE to understand component ports, port interface types, platform dependencies, and associated component metadata. One of the goals of the **Alexandria** component repository described in Section 4 is to provide a common repository for component descriptions for use by tools such as a PSE.

Unlike industry approaches, scientific component technology must support communicating parallel components. In most high-performance applications, components will communicate within the same memory address space, although the components themselves may be distributed across processor memories in a SPMD fashion. Some applications, however, will span multiple parallel computers. For example, a large simulation running on thousands of processors may be connected to a visualization component running on a small visualization engine with a few tens of processors. In this case, the component architecture must support some form of parallel data redistribution. A number of researchers have addressed this issue for certain limited classes of data types. Both PAWS [5] and CUMULVS [16] support parallel redistribution of arrays and other predefined data items such as particles or simple unstructured meshes. PARDIS [15] and Cobra [22] support distributed sequences and arrays in CORBA. We and other members of the CCA working group are researching

approaches for extending this work to more general scientific objects, but that work is preliminary and beyond the scope of this paper.

3. LANGUAGE INTEROPERABILITY TECHNOLOGY

Computational scientists developing large simulation codes often face difficulties due to language incompatibilities among various software libraries. Scientific software libraries are written in a variety of programming languages, including `Fortran`, `C`, `C++`, or a scripting language such as `Python`. Language differences often force software developers to generate mediating "glue" code by hand. In the worst case, computational scientists may need to re-write a particular library from scratch or not use it at all.

We have developed a tool called **Babel** that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation, description, and distribution of language independent software libraries. In the following sections, we describe our interoperability approach, the **Babel** tool architecture, and an example of using **Babel** in a parallel linear algebra software library.

3.1. SCIENTIFIC IDL

Babel addresses the language interoperability problem using Interface Definition Language (IDL) techniques [12, 19]. An IDL describes the calling interface (but not the implementation) of a particular software library. IDL tools use this interface description to generate "glue code" that allows a software library implemented in one supported language to be called from any other supported language. We have designed a Scientific Interface Definition Language (SIDL) that addresses the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to `Java`.

As illustrated in Figure 1, SIDL bears a close resemblance to CORBA and `Java`. The `package` keyword introduces a new namespace. A namespace may contain a class, interface, enumerated type, or another package. Classes and interfaces contain methods. The methods in an interface are abstract; that is, they are not implemented by the interface. As in CORBA, `in`, `out`, and `inout` modify method arguments and denote the direction of information transfer. SIDL also supports `Javadoc`-style documentation comments, which may be used to automatically generate browsable documentation (see the **Alexandria** discussion in Section 4).

```
version hypre 1.0;

/**
 * A SIDL type description for the <em>hypre</em> library.
 */
package hypre {

   /**
    * <code>Vector</code> represents a mathematical vector.
    */
   interface Vector {
      Vector clone();
      void scale(in double a);
      double dot(in Vector x);
      void axpy(in double a, in Vector x);
      int getGlobalDimension();
      int getLocalDimension() local;
   }

   /**
    * An <code>Operator</code> maps one vector into another vector.
    */
   interface Operator {
      void apply(in Vector x, out Vector y);
   }

   /**
    * This interface represents the class of linear mappings.
    */
   interface LinearOperator extends Operator {
   }

   /**
    * <code>StructVector</code> is a vector for structured grids.
    */
   class StructVector implements-all Vector {
      array<int> getGhostCellWidth();
   }

   /**
    * The structured matrix class implements all operator functions.
    */
   class StructMatrix implements-all Operator {
      // methods used to build a structured matrix omitted
   }
}
```

Figure 1 A simplified SIDL interface description for portions of the *hypre* software library described in Section 3.3.

The following sections provide additional details concerning some of the more unique characteristics of the SIDL interface definition language.

3.1.1 Symbol Versioning. In SIDL, every package, enumerated type, class, and interface is assigned a particular version number. Every SIDL description begins with one or more `version` statements. Each `version` statement contains a package name and an arbitrary version string consisting of a sequence of integers separated by periods. All symbols within a package share its version number. For example, the `version` statement on the first line of Figure 1 states that all symbols defined in the `hypre` package will be version 1.0 of that symbol. A `version` statement is required for every new outermost package defined in a SIDL description. A `version` statement may also be used to give an explicit version number for resolving external symbols referenced in a SIDL description. If a version is not specified for a particular external symbol, then the most recent version of that symbol is used.

Symbol versioning is an important consideration for the development of community-wide standards and specifications. Consider a standards committee that releases version 1.0 of a particular specification. Components will be written to and implement that version of the standard. When the committee releases version 2.0 of the specification, some components will immediately implement the new standard, whereas others will take longer. Versioning removes ambiguity about which version of the specification a particular component implements.

3.1.2 Import. Like `Java`, SIDL supports a type of `import` statement. The `import` statement adds the specified package name to the symbol resolution path. For example, a SIDL description that references symbol `Vector` in package `hypre` could either use the fully qualified name `hypre.Vector` or begin with "`import hypre`" and then simply use the name `Vector` (assuming, of course, that another `Vector` did not already exist in that name scope). External symbol references are resolved by searching an associated symbol repository, either a file repository or a web-enabled repository such as **Alexandria**.

3.1.3 Inheritance Model. The SIDL inheritance model is similar to that of `Java`. SIDL supports both interfaces and classes. The methods in an interface are abstract and thus not implemented by that interface. The methods in a class may be either abstract or implemented by that class. SIDL supports multiple inheritance of interfaces but single implementation inheritance of classes. An interface may extend other interfaces. A class may implement many interfaces but extend only one other class. This inheritance model simplifies the **Babel** implementation and removes the diamond implementation

inheritance ambiguity associated with C++. Like COM [12], all classes and interfaces implicitly inherit from a common base interface that provides reference counting and simple query interface capabilities.

Based on suggestions from our users, we have augmented the Java inheritance syntax with an implements-all keyword, which declares that the associated class implements all of the methods in the specified interface. This keyword is equivalent to using the implements keyword and repeating the definition of all interface methods in the class body. The implements-all shorthand is cleaner and more closely reflects the way many of our users think about designing scientific libraries. They typically define abstract interfaces that describe the desired functionality and then combine those interfaces together into classes and components that implement that functionality.

3.1.4 Arrays. SIDL supports the style of dynamically-sized, dense, multi-dimensional arrays that are common in scientific applications. Existing IDLs such as CORBA [19] support only dynamically-sized, one-dimensional arrays (a CORBA sequence) and statically-sized, multi-dimensional arrays. Dense arrays consist of one physical segment of memory that can be accessed efficiently by an optimizing compiler. Such arrays are common in the scientific community due to its Fortran heritage and because dense arrays offer better access performance than "array of array" implementations.

3.1.5 Parallelization Support. We have just begun to develop support for parallel data redistribution in the **Babel** tools. Therefore, the following discussion should be considered preliminary, although it does indicate our basic approach. SIDL currently supports parallel communication directives that describe method behavior in a parallel execution environment. For example, the local method modifier in class Vector of Figure 1 indicates that the getLocalDimension method is valid only when invoked on an object in the same memory address space. For this method, the number of local vector elements owned by a particular processor has no meaning for a Vector object distributed across a different set of processors.

Unlike PARDIS [15] and Cobra [22], we do not intend to add data distribution directives to the SIDL language. We do not believe that static IDL data distribution directives will be sufficient to describe the dynamic complexity and wide range of parallel objects used in scientific computing. Instead, we plan to use run-time data descriptions of data objects. Distributed parallel objects will be required to support one of a set of data distribution interfaces through which the object describes its internal data distribution state. The **Babel** runtime will use that information to manage data redistribution during method invocations. We feel this approach is more appropriate for sophisticated data decompositions that change during the course of a simulation.

3.2. BABEL TOOL ARCHITECTURE

The **Babel** tool suite consists of a number of separate pieces: a SIDL parser, a code generator, a small run-time support library, and the **Alexandria** component repository. Currently, **Babel** supports `Fortran 77`, `C`, and `C++`; we plan to develop support for `Java`, `Python`, `Fortran 90`, and `MATLAB` in the following year.

The **Babel** parser, which is available either at the command-line or through the **Alexandria** web interface, reads SIDL interface specifications and generates an intermediate XML [28] representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a repository or a problem solving environment. XML interface descriptions are stored either in a local file repository or on the web using **Alexandria**. The vision is that a scientist downloading a particular software library from the **Alexandria** component repository will receive not only that library but also the required language bindings generated automatically by the **Babel** tools.

The **Babel** code generator reads SIDL XML descriptions and automatically generates glue code for the specified software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space and, eventually, across memory spaces for distributed objects. The code generators create four different types of files: stubs, skeletons, **Babel** internal representation, and implementation prototypes. The **Babel** internal object representation created by the code generators is similar to that used by COM [12], CORBA's Portable Object Adaptor [19], and scientific libraries such as PETSc [2]. The internal object representation is essentially a table of function pointers, one for each method in an object's interface, along with other information such as internal object state data, parent classes and interfaces, and **Babel** data structures. Stub and skeleton code translates between the calling conventions of a particular language and the internal **Babel** representation. The code generators also create implementation files that contain function prototypes to be filled in by the library developers. To simplify the task of library writers, we have added automatic `Makefile` generation as well as a "code splicing" capability that preserves old edits during the regeneration of implementation files after modifications to the SIDL source.

3.3. TECHNOLOGY DEMONSTRATION IN HYPRE

In collaboration with members of the *hypre* development team, we have integrated some of the **Babel** language interoperability technology into *hypre* [9]. The *hypre* library is a suite of parallel scalable linear solvers and preconditioners implemented in `C` with MPI. There were four primary goals of this collaboration. First, the **Babel** team wished to demonstrate the technology and

get feedback from library developers. Second, the *hypre* project needed automatically generated `Fortran` bindings that would track changes in the library. Previously, a number of different `Fortran` bindings were developed by various users but fell into obsolescence with new changes to the *hypre* source. Third, the *hypre* team wanted to explore new design options using object-oriented and component-based software techniques, but the team had no desire to generate and support the necessary object-oriented infrastructure by hand. Finally, *hypre* developers wanted to integrate software developed by other groups who had written code in `C++` and `Fortran`.

The collaboration began by identifying key parts of *hypre* and developing an object-oriented design in SIDL for the primary *hypre* objects. For the most part, existing *hypre* implementations were wrapped using glue code generated by the **Babel** tools. In spite of this additional intermediate glue code, parallel runs with both `Fortran` and `C` drivers indicate that **Babel** overheads are too small to measure accurately.

The developers of *hypre* identified a number of advantages to using **Babel** for their scientific software library in addition to the obvious advantage of language interoperability. Developers found that SIDL was a convenient specification description language for the design of scientific libraries because it eliminated unnecessary implementation details and forced them to focus on the object-oriented design of the library. They felt that SIDL was relatively easy to master, although some were new to object-oriented design and object-oriented languages. Furthermore, *hypre* developers noticed that they could eliminate redundant code by taking advantage of polymorphism. For example, the previous *hypre* library contained a four different preconditioned conjugate gradient routines, each written for a particular type of preconditioner data structure. Through the use of polymorphism enabled by **Babel**, they were able to reduce the number of routines to one. Finally, the *hypre* developers were able to exploit object-oriented design in `C`, which has no object-oriented support, using the automatically generated **Babel** code.

4. THE ALEXANDRIA REPOSITORY

The **Alexandria** repository was designed and built to facilitate the adoption of component technology for high-performance scientific simulation software. Our goal was to provide a network service where component developers can publish their software and interface definitions and where application developers can find and download components and the associated language bindings. The system was intended to have a user interface to support human and machine clients.

Alexandria provides a hierarchically organized collection of software packages uploaded by component developers, a fuzzy search capability, an interface

definition browser, and a web user interface to the **Babel** language interoperability tool. For machine clients, **Alexandria** provides a repository of XML interface definitions and will hold a repository of shared libraries which implement particular interfaces to enable dynamic graphical application builders or other development tools.

We chose to implement a web application (i.e., a web server with dynamic content managed by a program) to achieve these goals and features. A web application can provide a sophisticated and friendly user interface designed for human clients and a simple, feature-rich interface for machine clients. By using web technologies, we make the repository's services available to the largest possible network audience; any contemporary web browser can access **Alexandria**. Machine clients can use standard network libraries to access the repository. Other network approaches would require installation of special purpose clients or more elaborate machine clients thereby decreasing the potential audience for the service. The HTTP protocol provides all the transaction types necessary for the repository: uploading files and other information from a user interface form and downloading content. The transactional nature of the web makes the user interface less interactive than a native application, but the benefits of the web interface seem to outweigh this deficiency.

As shown in Figure 2, **Alexandria** uses a three-tiered architecture: a web browser based user interface, a web server with `Java` servlets [11] and JavaServer Pages [21], and a JDBC [26] connection to an SQL backend. The web server delegates HTTP messages for certain URLs to `Java` servlets, and the servlet provides the content or an error response. A servlet is a `Java` class that implements a standard interface or overrides methods inherited from a standard base class. The servlet can use all the features of the `Java` platform in generating its response. JavaServer Pages is a convenient, dynamic way to generate a servlet which usually combines HTML with embedded `Java` code to provide the dynamic content.

The **Alexandria** application consists of five subsystems: an access control system, an inexact string matching package, a hierarchy management system, a content package, and an interface to **Babel**. The access control system manages user accounts and provides several different levels of access to the system: administrator, trusted user, normal user and world. The inexact string matching package is a Java implementation of the algorithm from `agrep` [30].

The hierarchy management system provides cataloging, uploading and downloading features. Unlike a normal file system, the hierarchy can hold files with the same name in a common directory as long as they have different version numbers. The expectation is that over time a project will issue multiple versions of individual files.

The content scanning package checks material provided by users to see if it is "safe content." A responsible web server that receives content from users and

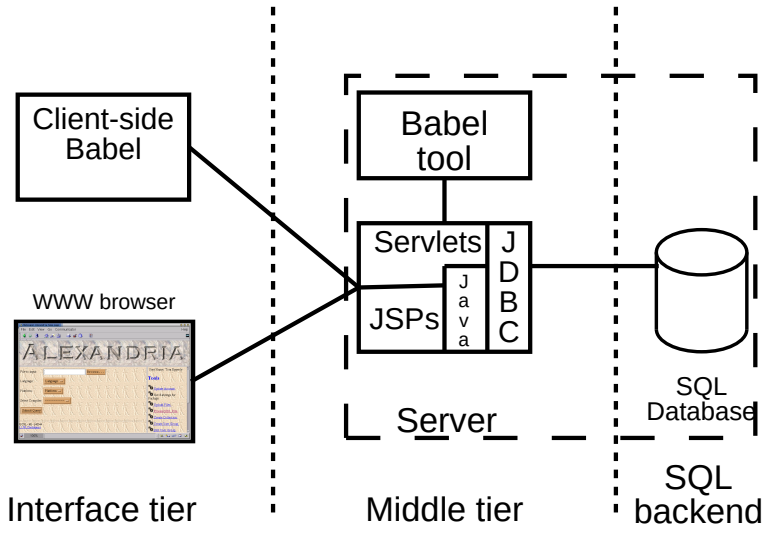

Figure 2 Alexandria architecture

then presents that content back to other users must verify that the user provided material does not contain hostile scripts. Rather than trying to characterize and detect hostile content, **Alexandria** tests user provided content against an XML DTD that contains a safe subset of XHTML 1.0 [27]. A validating XML parser is used to determine if user provided content is safe. If the material does not validate, all the mark-up directives are transformed so they will be interpreted as plain text rather than as mark-up directives.

The interface to **Babel** subsystem provides language bindings for a SIDL file to users. The user's SIDL file is uploaded to the web server, the web server runs **Babel** on the file, the results are packaged in a `TAR` file, and then the user is given the chance to download the file. This saves users from having to install **Babel** and a Java virtual machine on their local machine.

Alexandria maintains a repository of XML type information. Users with sufficient access can translate the SIDL file into an equivalent XML representation and upload the XML representation to the repository. Once it is in the repository, anyone running **Babel** can use the XML information from **Alexandria** rather than having to explicitly download all the needed SIDL files. In addition, the web server provides high quality interface documentation to web browser by applying XSLT [29], a evolving standard for translating XML into HTML or other markup languages.

5. CONCLUSIONS

In this paper, we have described two pieces of a component technology architecture for scientific computing. **Babel** is a language interoperability tool that uses the SIDL interface description language to describe component interfaces and to generate code that mediates differences between programming languages. **Alexandria** is a web-enabled component repository that provides a browsable software library, automated access to SIDL type information, and web access to the **Babel** code generators.

Obviously, much work remains in developing production-quality component technology for the scientific computing community. Members of the Common Component Architecture working group have made some initial progress in this direction and have drafted a proposal that covers common behavior standards for components [1]. A number of interesting open research questions remain in extending current parallel data redistribution approaches [5, 15, 16, 22] to arbitrary data components.

Acknowledgments

We would like to thank Andrew Cleary, Jeff Painter, and Cal Ribbens for integrating the **Babel** language interoperability technology into the *hypre* library and for their many useful suggestions. We would also like to thank members of the Common Component Architecture forum for numerous in-depth conversations about component technology for scientific computing.

References

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Curfman-McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings the Eighth International Symposium on High Performance Distributed Computing*, 1999. See `http://z.ca.sandia.gov/~cca-forum`.

[2] S. Balay, W. D. Gropp, L. Curfman-McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997. See `http://www.mcs.anl.gov/petsc`.

[3] D. Beazley. *SWIG Users Manual.* See `http://www.swig.org`.

[4] D. M. Beazley and P. S. Lomdahl. Building flexible large-scale scientific computing applications with scripting languages. In *The 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[5] P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of*

the High Performance Distributed Computing Conference, 1998. See `http://www.acl.lanl.gov/paws`.

[6] R. Bramley, K. Chiu, C. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of the High Performance Distributed Computing Conference*, 2000. See `http://www.extreme.indiana.edu/ccat`.

[7] D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/Overture`.

[8] K. G. Budge and J. S. Peery. Experiences developing ALEGRA: A C++ coupled physics framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[9] E. Chow, A. J. Cleary, and R. D. Falgout. Design of the *hypre* preconditioner library. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[10] J. Cummings, J. Crotinger, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T. Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.acl.lanl.gov/pooma`.

[11] J.D. Davidson and D. Coward. *Java Servlet Specification, v2.2*. See `http://java.sun.com/products/servlet/`.

[12] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[13] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Component architectures for distributed scientific problem solving. In *IEEE Computational Science and Engineering*, 1998.

[14] R. Hornung and S. Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/SAMRAI`.

[15] K. Keahey and D. Gannon. PARDIS: A parallel approach to CORBA. In *Proceedings of the Sixth IEEE Symposium on High Performance Distributed Computation*, 1997.

[16] J. Kohl and P. Papadopoulos. Efficient and flexible fault tolerance and migration of scientific simulations using CUMULVS. In *Second SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998. See `http://www.epm.ornl.gov/cs/cumulvs.html`.

[17] Los Alamos National Laboratory. *SILOON: Scripting Interface Languages for Object-Oriented Numerics.* Available at `http://www.acl.lanl.gov/siloon`.

[18] Microsoft Corporation. *Microsoft .NET Platform.* Available at `http://www.microsoft.com/net`.

[19] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Available at `http://www.omg.org/corba`.

[20] S. G. Parker, D. M. Beazley, and C. R. Johnson. *The SCIRun Computational Steering Software System*. E. Arge, A.M. Bruaset, and H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhauser Press, 1997.

[21] E. Pelegrí-Llopart and L. Cable. *JavaServer Pages Specification: Version 1.1*. See `http://java.sun.com/products/jsp/`.

[22] T. Priol, C. René, and G. Alléon. Programming SCI clusters using parallel CORBA objects. In *SCI-based Cluster Computing.* Springer Verlag, 1999.

[23] Sun Microsystems. *Enterprise JavaBeans Server-Side Component Architecture*. See `http://java.sun.com/products/ejb`.

[24] Sun Microsystems. *JavaBeans Component Architecture Documentation.* See `http://java.sun.com/products/javabeans/docs`.

[25] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[26] S. White and M. Hapner. *JDBC 2.1 API*. Sun Microsystems, Inc., 1999. Available at `http://java.sun.com/products/jdbc/`.

[27] World Wide Web Consortium. *The Extensible HypreText Markup Language*. See `http://www.w3c.org/TR/xhtml`.

[28] World Wide Web Consortium. *Extensible Markup Language (XML)*. See `http://www.w3c.org/XML`.

[29] World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*, 1999. Available at `http://www.w3.org/TR/xslt/`.

[30] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551

Preprint
UCRL-JC-140349

Divorcing Language Dependencies from a Scientific Software Library

S. Kohn, G. Kumfert, J. Painter, and C. Ribbens

This article was submitted to the
10th SIAM Conference on Parallel Processing, Portsmouth, VA,
March 12-14, 2001.

March 12, 2001

Divorcing Language Dependencies from a Scientific Software Library*

S. Kohn†, *G. Kumfert*†, *J. Painter*†, *and C. Ribbens*‡

1 Introduction

In scientific programming, the never-ending push to increase fidelity, flops, and physics is hitting a major barrier: scalability. In the context of this paper, we do not mean the run-time scalability of code on processors, but implementation scalability of numbers of people working on a single code. With the kinds of multi-disciplinary, multi-physics, multi-resolution applications that are here and on the horizon, it is clear that no single code group — nor any single organization — has all the required expertise or time available to independently create all of the software needed to solve today's cutting-edge computational problems.

Scientific programming libraries have alleviated some of this pressure in the past, but scaling problems are becoming increasingly apparent. The upshot of software libraries has been that different code groups in different organizations can bring their expertise to bear on particular sub-problems. Unfortunately, different groups and different organizations also bring with them implicit dependencies on different software development platforms, different programming languages, and different conceptual models of the problem decomposition — all of which must be resolved if the libraries they produce are to be useful in a final application. The good news is that scientific computing is not alone in these software scalability problems and several industry solutions have proven successful. The bad news

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. UCRL-JC-140349

†Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory

‡Department of Computer Science, Virginia Tech & on sabbatical at the Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory

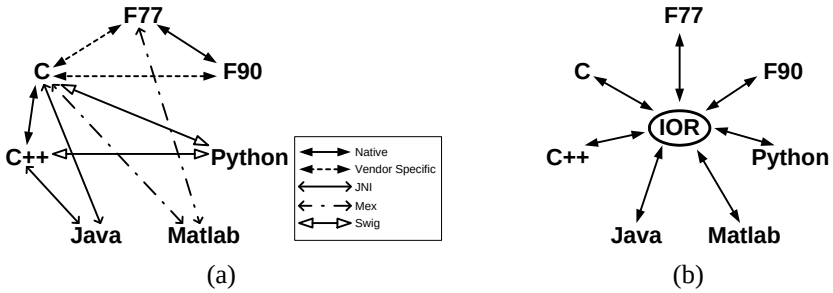

Figure 1. *Language interoperability without (a), and with (b) IDL techniques*

is that scientific computing is different enough in its nature for an "off-the-shelf" solution from industry to not quite fit the scientific computing domain.

This paper describes the ideas, process, and results of the first year in an ongoing collaboration between members of the Components Project and the Hypre Project in the Center for Applied Scientific Computing (CASC) in Lawrence Livermore National Laboratory. The Components Project has developed a tool called Babel that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation and distribution of language independent software libraries. Hypre is a parallel, scalable scientific library of linear solvers and preconditioners. By using Babel tools on Hypre in this collaboration, we found that Babel enables better software design and is an effective tool for producing language independent scientific software libraries at a negligible performance overhead.

2 SIDL

It is already very common in scientific computing to have libraries written in different languages interoperate. Consider the common case of Basic Linear Algebra Subroutines (BLAS) written in Fortran77 and invoked from C/C++. Although vendors have provided custom solutions for this problem for years, this solution has scaling problems for general libraries. First, BLAS are often tuned specifically for the target architecture. Second, glue code has to be written for C/C++ to call the Fortran subroutines. Third, the Fortran77 standard does not define the binary calling interface between C/C++ and Fortran77, so the wrappers are also vendor specific.

Many programming languages can call other languages, but only on a pairwise basis. These pairs often require significant effort (meaning wrappers or "glue code"), are not guaranteed to be portable, and may require special interconnect technology. This is illustrated in Figure 1(a). For instance, Matlab can be coaxed to run an external library written in C, but to do so means writing special Mex-Files. Getting Matlab to run a Python script natively is another matter entirely.

In large, multidisciplinary scientific applications, we are increasingly observing a need for truly language independent pieces of software. One can easily envision an application with Java or Tcl/Tk graphical displays, Python scripts driving the highest levels of

logic, Fortran linear algebra routines, solvers written in C, and the adaptive mesh refinement and time-stepping management infrastructure written in C++. Such an application would be almost impossible using the technology represented in Figure 1(a).

This problem is addressed in industry using component technologies such as CORBA and COM. In both cases, language interoperability is achieved using Interface Definition Languages (IDLs).

The Components Project has designed a Scientific Interface Definition Language (SIDL) that addresses the particular needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallelization attributes and communication directives that are required for general parallel distributed data structures, all of which are lacking from industry IDLs. SIDL also provides other features that are generally useful but not necessarily related to scientific computing, such as an object-oriented inheritance model similar to Java, name space management, and interface versioning.

SIDL is not a "lowest-common-denominator" solution between programming languages. SIDL supports full object-oriented programming, even in non object-oriented languages. It implements reference counting and dynamic type casting, even in Fortran77 which has no aliasing and limited type casting through equivalence statements.

3 Babel

The Babel tool suite takes the SIDL descriptions and a language/platform description of a software library and generates all of the glue-code on demand. It consists of a number of interrelated pieces: a SIDL parser, a code generator, a small run-time support library, and a software repository. Currently, Babel supports Fortran77, C, and C++; efforts are underway to support Java, Python, Fortran 90, and Matlab.

The Babel parser, which is available either at the command-line or through a web interface, reads SIDL interface specifications and generates an intermediate XML representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a web-based repository or a GUI development environment. XML interface descriptions are stored locally or in a shared web-based software repository called Alexandria[1]. The vision is that a scientist downloading a particular software library from the repository will receive not only that library but also the required language bindings generated automatically by the Babel tools.

The Babel code generator reads XML files and generates glue code for linking from a software library to an *intermediate object representation* (IOR), and from the IOR to the application programmer's language of choice (see Figure 1(b)). This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space. The IOR used by the code generator is similar to that used by COM, CORBA's Portable Object Adaptor, or by scientific libraries such as PETSc [2, 3]. The IOR handles the virtual function dispatch for all the methods in an object's interface, maintains the object's state data, and manages some internal Babel data structures and metadata.

[1] Also developed in the Components Project, but beyond the scope of this paper.

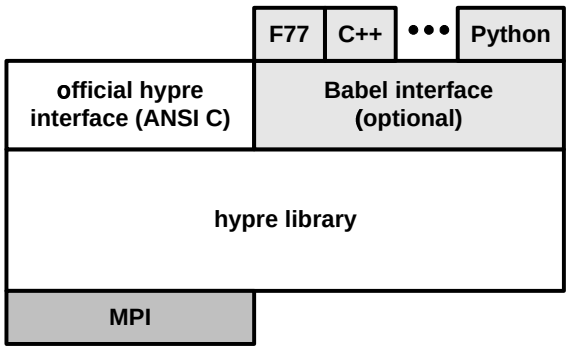

Figure 2. *The original vision of hypre and Babel*

4 Hypre-Babel Collaboration

Hypre [5] is a suite of scalable parallel linear solvers and preconditioners for the solution of large, sparse linear systems of equations on distributed-memory parallel computers. The primary algorithmic emphasis in Hypre is on robustness and scalable parallel performance. In addition, important design goals for the library include ease of use, flexibility, the rapid incorporation of new algorithms, and compatibility and interoperability with other similar libraries. These goals and emphases are driven by the needs of the most demanding scientific simulation codes, as typified by the U.S. Department of Energy's Accelerated Strategic Computing Initiative (ASCI).

The collaboration between Hypre and Babel began by identifying four primary goals and a vision of how the two projects interact. The four primary goals are:

1. The Babel team wanted to demonstrate the technology and get feedback from library developers.

2. The Hypre project had an immediate need for automatically generated Fortran bindings that would track changes in the library. Future needs for bindings to other languages (e.g., Python) was considered extremely likely. Previously, a number of different Fortran bindings were developed by various users on various platforms but fell into obsolescence with new changes to the Hypre library.

3. Hypre developers wanted to integrate software developed by other groups who had written code in C++ and Fortran.

4. The Hypre team wanted to explore new design options using object-oriented and component-based software techniques, but the team had no desire to generate and support the necessary object-oriented infrastructure by hand. This included a desire to participate in the Equation Solver Interface (ESI) working group [6], which requires working implementations to verify proposed designs.

The original vision of how Babel was to interface to Hypre is shown in Figure 2. Hypre makes a clear distinction between their "official" (meaning published and supported) pro-

```
interface Vector {
  int Clear();
  int Copy( in Vector x );
  int Clone( out Vector x );
  int Scale( in double a );
  int Dot( in Vector x, out double d );
  int Axpy( in double a, in Vector x );
};

interface Operator {
  int Apply( in Vector x, out Vector b );
};

interface LinearOperator extends Operator {
};

interface Solver extends LinearOperator {
  int GetSystemOperator( out LinearOperator op );
  int GetResidual( out Vector resid );
  int GetConvergenceInfo( in string name, out double value );
};

interface PreconditionedSolver extends Solver {
  int GetPreconditioner( out Solver precond );
};

interface RowAccess extends LinearOperator {
  int GetRow( in int row, out int size,
              out array<int,1> col_ind,
              out array<double,1> values );
};
```

Figure 3. *SIDL definition of some basic Hypre interfaces. Not all methods are shown.*

gramming interface in ANSI C and the library proper which was subject to more frequent change during the course of research. The original expectation was to supply an optional Babel interface to support other languages as they came on line.

Due to the overall size of Hypre, our initial focus was on designing and implementing a Babel interface for a representative subset of the library. We developed a SIDL file that matched the programming interfaces of this Hypre subset while adhering to SIDL's object model. We then generated the glue code between Hypre and the Babel IOR, and hand-edited the implementation details to finish the new language-independent library.

SIDL's object-model follows that of Objective-C and Java, using *classes* and *interfaces*. For C++ programmers, interfaces are similar to classes except that all methods are pure virtual, meaning they have no implementation. In this model, a class can inherit an implementation from only one class, but may inherit multiple interfaces. Figure 3 shows a SIDL definition of several key interfaces in the Hypre object hierarchy.

Table 1. *Runtime (in seconds) for a SMG multigrid solver on a* $40 \times 40 \times 40$ *structured mesh with a seven point stencil on ASCI-Blue Pacific*

	setup	solution
standard Hypre C interface	8.07	43.08
standard Hypre C interface	8.07	42.96
Babel-generated C interface	8.09	42.45
Babel-generated C interface	8.05	42.76

5 Results

We are very pleased and encouraged by the results of this collaboration between the two research groups. The performance and interoperability results were in line with expectations. Additionally there were some unexpected results that were very positive and constructive.

Negligible Runtime Overhead. Results of four runs of a standard Hypre test problem are reported in Table 1. The test problem uses Hypre's SMG multigrid solver on a Poisson equation in three dimensions, finite-differenced on a seven-point stencil, on a uniform $40 \times 40 \times 40$ structured mesh. The timings were measured using eight processors on two nodes of ASCI Blue-Pacific, a large system based on IBM RS/6000. The times reported are the sum of the times of the eight processors. Most of the manipulation through either set of interfaces is done during the setup phase. The solution phase is practically entirely within the Hypre library proper.

It is clear to see in this example that the overhead of using the Babel interface is well within the noise of the system. Moreover, it is reassuring to see that Babel can be added to existing MPI based SPMD code without ruining parallel performance.

Reduced Code Size Through Polymorphism. Some Hypre implementations proved to be unnecessary once the SIDL defined interfaces were available. For example, it was easy for the Hypre team to write generic implementations of common solvers. Given definitions of interfaces such as `Vector`, `LinearOperator`, and `RowAccess`, it is natural to implement Krylov solvers such as conjugate gradient and GMRES in terms of these interfaces. These solvers can then work with any concrete classes that implement the required interfaces. There is no longer a need to write and maintain multiple versions of common solvers, one for each matrix data type.

Originally, Hypre included eight implementations of PCG (preconditioned conjugate gradient), some of them almost identical except for how they handled the matrix-vector multiply, because of data-structure differences. To take advantage of Babel's polymorphism capabilities, we coded a PCG solver which exclusively used the Babel interface to manipulate vectors. We have Babel interfaces for two vector types so far, so this PCG solver effectively replaces two separate implemenations in the Hypre library. Likewise Hypre developers have similarly been able to reduce the number of GMRES (generalized minimal residual) solvers.

Table 2. *Runtime (in seconds) for a SMG multigrid solver on a* $10 \times 10 \times 10$ *structured mesh with a seven point stencil on Sun Sparcstation Ultra 10*

	setup	solution
standard Hypre C interface	0.14	0.26
Babel-generated F77 interface	0.14	0.27

Hypre developers involved in this collaboration feel that using Babel will allow users to get the benefits of object-oriented design without requiring object-oriented languages such as C++, which is much less portable than C.

Automatic Language Bindings. Babel was used to generate a Fortran interface to the same Hypre library (which is written in ANSI C). We ran some of the same test problems from a Fortran driver and obtained the same numerical results on a Sun workstation. This successfully demonstrated a key goal to the Hypre developers. Previous Fortran interfaces have required frequent maintenance and lacked portability.

We present in Table 2 some runtime results that again show no real difference between the performance through the Hypre and Babel interfaces. This was done on a single processor Sun workstation using a smaller version of the problem in the previous section.

Hypre developers involved in this collaboration are confident that an application code written in terms of a particular set of interfaces could use any solver or library that implements those interfaces, with virtually no change to the application code. Users could easily experiment with using different solver libraries by simply replacing one library's implementation of the required interfaces with another library's implementation.

Explore New Design Options. In addition to the basic Hypre objects defined by the interfaces shown in Figure 3, a second set of interfaces, called *builder* interfaces, were developed and plays a role of increasing importance. A builder interface is a set of methods for constructing one or more basic objects and follows the Builder design pattern [7]. These builders have no concrete analog in the Hypre library and are exclusively available through the Babel interface. A major benefit of the builders is that users are prevented from accessing partially constructed datastructures.

The most interesting examples are the `MatrixBuilder` and `SolverBuilder` interfaces. A `MatrixBuilder` can be thought of as a particular user interface through which users define problems. Each `MatrixBuilder` is accompanied by a `VectorBuilder` for building compatible vectors. A `SolverBuilder` is used to set the components and parameters that define a `Solver`. Partial SIDL definitions of builder interfaces are given in Figure 4.

SIDL as a Design Language. To generate interface code Babel requires a SIDL file defining the interfaces. This forced the Hypre developers to consider the user interface as a separate issue from the implementation, and provided an automated mechanism to keep

```
interface MatrixBuilder {
  int SetMap( in Map map );
  int Setup();
  int GetConstructedObject(out LinearOperator obj);
};

interface StructuredGridMatrixBuilder extends MatrixBuilder {
  int Start( in StructGrid grid, in StructStencil stencil,
             in int symmetric, in array<int,1> num_ghost );
  int SetValue( in array<int,1> where, in double value );
  int SetBoxValues( in Box box, in array<int,1> stencil_indices,
                    in array<double,1> values );
};

interface IJMatrixBuilder extends MatrixBuilder {
  int Start( in MPI_Com com, in int m_global, in int n_global );
  int SetLocalSize( in int m_local, in int n_local );
  int SetRowSizes( in array<int,1> sizes );
  int InsertRow( in int n, in int row,
                 in array<int,1> cols,
                 in array<double,1> values );
};

interface SolverBuilder {
  int Start( in MPI_Com comm );
  int SetParameterDouble( in string name, in double value );
  int SetParameterInt( in string name, in int value );
  int SetParameterString( in string name, in string value );
  int Setup( in LinearOperator A, in Vector b, in Vector x );
  int GetConstructedObject( out Solver obj );
};

interface PreconditionedSolverBuilder extends SolverBuilder {
  int SetPreconditioner( in Solver precond );
};
```

Figure 4. *Examples of Builder interfaces in Hypre. Not all methods are shown.*

the code consistent with the user interface design. There was no opportunity to clutter the interface with quick, one-time hacks. The result was a more stable and predictable user interface.

The simplicity of the SIDL file made it the most convenient language for Hypre developers to use to discuss user interface design. We could limit our discussion to pure interface issues while remaining confident that whatever we came up with would be practical. SIDL was an easy language to pick up and (unlike UML) was easy to write up in email and send to collaborators.

Improvements to SIDL. The Hypre interface project also provided useful feedback to the Babel project. Our experience with practical use of Babel led to several features and tools which now make Babel easy to use. One common mistake that was made was confusion over how to make a concrete class, i.e. one for which all the inherited virtual functions have an implementation to handle the calls. It was easy for classes to inherit a lot of interfaces, and the writer of the SIDL file to forget to add a single method signature that was supposed to be implemented only to find that Babel created an abstract, not concrete class.

To correct the situation, SIDL was modified in two ways. First, the keyword "abstract" was added to classes that may have unimplemented methods. If a method is left unimplemented and the class is not declared abstract, there is an error. Additionally, the keyword "implements-all" was added. If a class inherits an interface through a regular implements directive, it overrides only those methods explicitly mentioned in the class definition. If the interface is inherited through an "implements-all" directive, all the methods of the interface are expected to be overridden by the class and writing the method call in the class definition becomes redundant.

Improvements to Babel Tools. Based on observing the use of the Babel tools and interviews with the Hypre developers involved, two features were added to improve usability: automatic makefile generation, and preservation of user edits to generated code.

Even on a small SIDL file, the Babel tools can generate a surprising number of header and source files, often in various languages. The Babel code generators were modified so that a makefile fragment is generated in each directory where code is generated. These makefile fragments define macros that list the relevant filenames and are suitable for inclusion into larger makefiles.

In addition to the glue code that the Babel tools generate, they also generate so called *Impi* files with empty function bodies. Developers of new libraries may want to build their implementation directly in these files, but developers of legacy libraries use this as a place to simply dereference pointers and call their own code. We added functionality to the Babel tools so that if the SIDL file was changed incrementally, these edits to the Impl files are preserved. This improvement has saved Hypre developers a significant amount of cut-and-paste.

Revised Hypre Architecture. At the end of one year of a Hypre-Babel collaboration, a new vision is emerging about the Hypre architecture as shown in Figure 5. In this new design, the Hypre library will depend on the Babel runtime library to provide object-oriented support throughout the entire hypre library, not just the Babel interfaces. Additionally, all the published interfaces, including the ANSI C interface will be provided with Babel.

The design in Figure 5 represents a major shift in the Hypre library and has yet to be finally decided. The Babel developers are particularly pleased that though this collaboration, Hypre developers have developed so much enthusiasm for Babel tools.

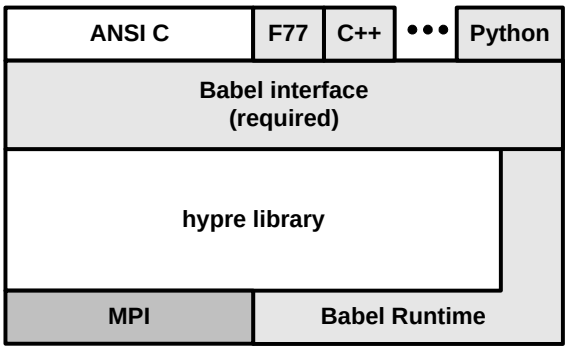

Figure 5. *The revised vision of hypre and Babel*

6 Conclusions and Future Work

Babel did the language interfacing job it had been designed for, at no cost to the Hypre user and great advantage to Hypre developers. The Hypre-Babel collaboration led to improved codes, and methodologies for both groups.

In the long term, Hypre plans to increase its reliance on the Babel tools and may eventually be distributed with pregenerated interfaces for several languages and platforms, and a BabelLite runtime library. In this configuration, it is entirely possible that the users of the library don't even have to be aware that they are using Babel as well. Members of the Hypre team also plan to continue participation in the Equation Solver Interface (ESI) [6] working group, developing standards for linear solver interfaces.

Babel continues to mature. Work is constantly being done to support additional languages and platforms. Much of the current research within the Components Project at LLNL is focused on handling parallel remote method invocations and data redistribution in a language independent manner. The Components Project also maintains close ties to a larger, grass-roots initiative called the Common Component Architecture (CCA) Forum [1, 4]. The goal of the CCA is to bring modern component technology to scientific computing. The Babel tools are targeted to provide the language independence to the CCA.

As scientific applications become more interdisciplinary the need for interoperability between different libraries and among pieces from different libraries becomes even more important. An important question is how a Babel/SIDL skin can easily be wrapped around existing libraries. The Hypre developers feel that if the existing library was reasonably well organized (even if not using an explicit OO language) the effort is reasonable, the runtime costs negligible, and the potential payoff in increased interoperability huge.

Bibliography

[1] R. ARMSTRONG, D. GANNON, A. GEIST, K. KEAHEY, S. KOHN, L. MCINNES, S. PARKER, AND B. SMOLINSKI, *Toward a common component architecture for high-performance scientific computing*, in Eighth IEEE Int'l Sym. on High-Performance Distributed Computing, Redondo Beach, CA, August 1999. also Lawrence Livermore National Laboratory technical report UCRL-JC-134475, June 1999.

[2] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Petsc: The portable extensible toolkit for scientific computing*. `http://www.mcs.anl.gov/petsc`.

[3] ——, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, Birkhauser Press, 1997, pp. 163–202.

[4] *Common Component Architecture (CCA) Forum* . `http://z.ca.sandia.gov/~cca-forum`.

[5] E. CHOW, A. CLEARY, AND R.FALGOUT, *Design of the hypre preconditioner library*, in Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. E. Henderson, C. R. Anderson, and S. L. Lyons, eds., SIAM, Philadelphia, PA, 1999, pp. 106–116.

[6] *Equation Solver Interface (ESI) Working Group* . `http://z.ca.sandia.gov/esi`.

[7] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional Computing Series, Addison Wesley Longman, 1995.

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551

Preprint
UCRL-JC-148723

A Component Architecture for High-Performance Computing

D. E. Berhholdt, W. R. Elwasif, J. A. Kohl, and T. G. W. Epperly

This article was submitted to the
Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL-02), New York, NY, June 22, 2002.

U.S. Department of Energy
Lawrence Livermore National Laboratory

June 22, 2002

A Component Architecture for High-Performance Computing*

David E. Bernholdt, Wael R. Elwasif, and James A. Kohl
{*bernholdtde,elwasifwr,kohlja*}*@ornl.gov*
Computer Science and Mathematics Division
Oak Ridge National Laboratory
P. O. Box 2008
Oak Ridge, TN 37831-6367 USA

Thomas G. W. Epperly
tepperly@llnl.gov
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P. O. Box 808
Livermore, CA 94551 USA

Abstract

The Common Component Architecture (CCA) provides a means for developers to manage the complexity of large-scale scientific software systems and to move toward a "plug and play" environment for high-performance computing. The CCA model allows for a direct connection between components within the same process to maintain performance on inter-component calls. It is neutral with respect to parallelism, allowing components to use whatever means they desire to communicate within their parallel "cohort." We will discuss in detail the importance of performance in the design of the CCA and will analyze the performance costs associated with features of the CCA.

1 Introduction

In some ways, high-performance scientific computing is a victim of its own success. The ability to simulate physical phenomena in a scientifically useful way leads to demands for more sophisticated simulations with greater fidelity and complexity. At the same time, the supercomputers on which such simulations are run grow ever more powerful, but simultaneously more complex. Obtaining maximum performance from modern supercomputers requires careful algorithm design, including management of multiple levels of the memory hierarchy. Combining the support of a range of modern supercomputer architectures with the increasing demands from the scientific side of the problem can lead to nearly unmanageable complexity in the software created for modern computational science.

The computer science community is exploring a variety of approaches to help alleviate some of the complexity of large-scale scientific software. Libraries or computational engines may be created with algorithms and/or algorithmic parameters optimized for the target computer system. While this approach focuses on performance issues, it may also provide some help with complexity, since in many cases algorithms are abstracted and parameterized to cover a range of computer systems. Domain-specific, high-level languages can greatly simplify the scientist's view of the scientific programming problem, but typically a scientist will rely on a large and complex infrastructure of libraries, computational engines, or generated code in more traditional programming languages. This approach shifts much of the complexity to the development of the libraries, engines, or tools, but once again to the extent these tools embody important (often domain-specific) abstractions and generalizations, they can further reduce the complexity of the overall software system.

*Research supported by the Mathematics, Information, and Computational Sciences Office, the Office of Advanced Scientific Computing Research, and the U. S. Department of Energy under contract no. DE-AC05-00OR22725 with UT-Battelle, LLC, and W-7405-Eng-48 with the University of California. LLNL report UCRL-JC-148723

Even with the benefit of such techniques, high-performance simulation software tends to be large and complex. Moreover, there is much "legacy" software in the scientific community which cannot, for technical or practical reasons (i.e., time or funding), be rewritten to the extent required to accommodate these high-level approaches. Consequently, it is valuable to look at other ways to help scientific programmers manage the complexity of their software systems.

One such approach which has become very popular and successful in other areas of computing, most notably the "business" and "internet" areas, is component-based programming. Components may be thought of as objects that encapsulate useful units of functionality and interact with other components only through well-defined interfaces. To the extent that these interfaces are specified in such a way as to be broadly useful to a community (i.e., scientific domain) rather than a specific program, the component approach can facilitate reuse and interoperability of code. Component-based applications are typically constructed by connecting the required components together in a software framework; creating a complex scientific application could become a matter of assembling components to express the "physics" of the problem and coupling them with numerical solvers and other components to form the complete software package. As component-based programming for scientific computing develops, we can anticipate that many of the components required for a given application would already have been created by experts in the relevant domains and made available through a component repository.

The Object Management Group's CORBA [19], Microsoft's COM and DCOM [29], and Sun's Enterprise JavaBeans [22] are examples of very popular component environments in the business and internet areas. Visualization systems such as Advanced Visualization System's AVS [31], OpenDX (derived from IBM's Data Explorer) [6], and VTK [10] also have a component flavor to them, with the connections between components typically representing data flow. Unfortunately, these environments do not address the requirements of high-performance scientific computing in various ways and have seen very limited use in scientific computing. Efforts by computational scientists to develop component environments [9, 13–16, 20, 26–28] have been mostly focused on specific problem domains, and tend to lack the generality and flexibility needed for use by a much broader user base.

In response to this situation, a grassroots effort was launched by researchers in several U.S. national laboratories and universities to create a component environment suited to the general needs of high-performance scientific computing. The resulting Common Component Architecture (CCA) [2] is now at the prototype stage and is being adopted by a wide variety of scientific computing projects.

2 An Overview of the Common Component Architecture

In the design of the CCA, a number of requirements were considered:

- **Performance:** It should impose a negligible performance penalty.
- **Portability:** It should support languages and platforms of interest in scientific computing.
- **Flexibility:** It should support a broad range of parallel programming paradigms, including SPMD, multithreaded, and distributed models.
- **Integration:** It should impose minimal requirements for existing software to be able to participate in the component environment.

The specification [1] developed by the CCA Forum defines:

- minimal required behavior for a CCA component,
- minimal required behavior for a CCA framework, and
- the interface between components and frameworks,

in such a way as to allow the above requirements to be satisfied, but, to the extent possible, without dictating specific solutions.

At the heart of the CCA is the concept of *ports*, through which components interact with each other and with the framework. Ports are merely interfaces that are completely separate from all implementation issues. They correspond to *interfaces* in Java, or *abstract virtual classes* in C++. CCA ports follow a uses-provides design pattern, so that each component must declare what ports it *uses* from others, and the ports for which it *provides* an implementation. This

typically occurs in the component's `setServices` method, which is invoked by the framework when the component is instantiated. `setServices` is the only method a CCA component is specifically required to implement.

A CCA framework is primarily a container for components being assembled into an application, which mediates the interconnection of ports. Its primary interaction with the component is through the `Services` interface, which allows components to register (used or provided) ports, and to get those ports for actual use. A design goal for the framework is to be able to cast as components even functionality that might be thought of as fundamental framework services. The details of such services are still evolving to some extent, but include things like event services, and "builder services," which provide the capabilities to load/unload components and to connect/disconnect ports. Currently both command-line and graphical means are provided to allow the user to assemble CCA applications.

During execution, when one component needs to use methods provided by a port on another component, it uses the `getPort` method of the `Services` interface to obtain a reference to the port, which can in turn be used to invoke the methods provided by the port. The using component calls `releasePort` when the port is no longer needed. The framework, through the `Services` object, mediates the invocation of methods provided by other components and is important in allowing the CCA to provide both high performance for "local" components and remote access in the case of distributed components; this will be explained in more detail below.

Another aspect of the CCA is the desire to make the use of components independent of their implementation language. To achieve this, we have adopted the Babel language interoperability tool [4]. A Scientific Interface Definition Language (SIDL) is used to generate the necessary glue code between languages. SIDL is also used by the CCA Forum to express the interfaces in the CCA standard. Babel currently supports C, C++, Fortran 77 (F77), Python, and client-side Java, with support for server-side Java and Fortran 90 planned.

The CCA specifications do not dictate implementation issues, such as exactly how calls are made from one component to another, but the model has been designed in such a way as to allow the implementation of highly efficient methods, preserving the innate performance of the environment. It is also worth noting that the specification says nothing specifically about parallelism. The basic philosophy in this matter is for the CCA environment to "stay out of the way" of parallelism. Performance matters are described in more detail in the next section.

3 Performance Considerations in the CCA

3.1 The CCA Framework

The CCA's uses-provides pattern for ports, in which the framework mediates the use of one component by another, is central to both the flexibility and performance of the model for inter-component calls. When a using component invokes `getPort`, the `Port` object returned by the framework might be a proxy for invocation of methods on a remote component in a distributed computing environment. In this case, it is up to the framework to marshal and unmarshal the arguments and make the remote invocation, and components on either end need not know they exist in a distributed environment. Of course, distributed computing is not usually considered to be a high-performance environment, and the CCA user creating the application is well advised to consider the frequency of use and the volume of data transfer in ports when setting up the application in a distributed environment.

In the case that both components are local, the `getPort` call might return a reference to the actual implementation. This is done, for example, in the prototype Ccaffeine framework [11], which focuses on supporting high-performance parallel CCA applications and is written in C++. Components (in the form of shared object libraries) are loaded into distinct namespaces within a single address space (process). The use of different namespaces ensures that the components cannot interfere with each other, and the framework is the only part of the environment which can "see" all components. Since components are all in a single address space, the framework can easily return a direct reference to the port's implementation from `getPort`. This is referred to as a *direct connection* environment, and allows one component to call methods on another with a cost equivalent to a C++ virtual function call — essentially, a lookup of the method in the component's function table, followed by invocation of that function.

Since `getPort` calls occur infrequently (they are required only once per used port), their cost is negligible. The overhead of the CCA framework is almost entirely due to the cost of inter-component calls relative to the equivalent calls in a native language environment. Since this overhead is on the order of the cost of a native language function call, it will not play a significant role in a great many inter-component calls — most scientific software is designed to put a reasonable amount of work in each function — nor will it effect *intra*-component calls. For those rare cases where the overhead imposed by the CCA framework is an unavoidable concern, we characterize the costs below.

3.2 Language Interoperability via Babel

With the use of Babel for language interoperability, as is now being introduced into the CCA environment, some additional overhead is introduced. Babel uses a C-based internal object representation (IOR) to provide the glue between different languages. In general, the overhead is roughly two subroutine calls. The client calls a stub routine that translates the arguments into C. The stub routine calls the skeleton routine which translates the arguments into the implementation language, and the skeleton calls the implementation. In some cases there is an additional overhead due to data conversions between languages (especially with character strings), and with existing code, the developer might need to insert an additional layer to adapt from the object-based representation used by Babel to the style of the existing code. As might be expected, when reasonable amounts of computation take place in the methods called via Babel, the overhead of the Babel system is not noticeable [21]. Obviously somewhat more care is required in using Babel with methods that might be called a large number of times and involve little work.

Babel is distinctive from other language interoperability tools because it provides bi-directional function calls. For example, a Python program can call a F77 subroutine that calls a C++ function that calls something implemented in Python. This flexibility comes at a cost. The software developer must write a SIDL file to describe the interfaces that will be accessible from multiple languages. There is also a runtime overhead that will be quantified below.

Babel was designed with the CCA in mind, but it can also be used without the CCA framework. Babel can be used to wrap legacy applications and libraries to provide a high-level, language-independent interface. The code wrapped by Babel can use native function calls in whatever the implementation language happens to be. When Babel is integrated with a CCA framework, the overhead of the "full" CCA environment is equivalent to the overhead of Babel — the framework's virtual function call is simply carried out in the Babel environment.

3.3 Parallelism

The final performance issue, and perhaps the most important for modern scientific computing, is that of parallelism. As noted, the CCA's primary approach to parallelism is staying out of the way of the parallelism built into the components. In a parallel environment, the CCA framework mediates interactions between components in the same process, just as it does in the sequential case. Interactions among parallel instances of a component in different processes (referred to as a *cohort*) are up to the developer of that component. Components may use whatever parallel communication environment they prefer (i.e., MPI [5, 18, 30], PVM [7, 17], Global Arrays [12, 23, 24], shared memory), and different components may even use different systems. The framework itself essentially does not know it is running in parallel, apart from the need in some cases to initialize the communication system — a dependence we plan to shift into a separate component shortly. Because the framework is, in effect, embarrassingly parallel, we will not concern ourselves with scalability measurements in this paper.

4 Performance Measurement Techniques

To characterize the performance overheads in CCA-based environments, we measured a variety of simple subroutine and function calls in native C, C++, and F77, in the C++-based Ccaffeine CCA framework, the C++-based omniORB CORBA environment, and in nine language combinations using Babel (C, C++, and F77 as calling language and called language). The functions were intended to illustrate the cost of calls passing a single variable of various data types. The complete list of functions and the environments in which they were tested is shown in Table 1. The functions are divided into several groups to simplify presentation and analysis of the results:

A: those for which Babel types map directly to native language types,

B: additional "simple" functions which show significantly different costs from group A in certain languages,

C: those requiring some measure of adaptation between languages in Babel and therefore show greater variation in cost, and

D: remaining functions, mostly those which are relevant only to object-oriented environments, such as Babel and (in some cases) C++.

Some of the function or arguments require a brief explanation:

Table 1: Measurements of function call overheads were obtained with a variety of arguments, corresponding to basic datatypes of the languages as well as additional special types introduced by Babel. This table details the languages and environments (Native, Babel, Ccaffeine, omniORB) in which each type was tested. Group designations are used to simplify the analysis.

		C		C++				F77	
Group	Function/Argument	N	B	N	B	C	O	N	B
A	Double	Y	Y	Y	Y	Y	Y	Y	Y
	Float	Y	Y	Y	Y	Y	Y	Y	Y
	Int	Y	Y	Y	Y	Y	Y	Y	Y
	Long	Y	Y	Y	Y	Y	Y	Y	Y
B	no arguments	Y	Y	Y	Y	Y	Y	Y	Y
	no args., returns double	Y	Y	Y	Y	Y	Y	Y	Y
C	Array	Y	Y	Y	Y	Y	Y	Y	Y
	Bool	Y	Y	Y	Y	Y	Y	Y	Y
	Complex (by reference)	Y		Y		Y		Y	
	Complex (by value)	Y	Y	Y	Y	Y	Y		Y
	Double Complex (by reference)	Y	Y	Y	Y	Y		Y	Y
	Double Complex (by value)	Y	Y	Y	Y	Y	Y		Y
	OrderedArray	Y	Y	Y	Y	Y	Y	Y	Y
	String (by reference)		Y	Y	Y	Y	Y		Y
	String (by value)	Y	Y	Y	Y	Y	Y	Y	Y
D	Char		Y		Y		Y		Y
	Interface		Y	Y	Y	Y	Y		Y
	no args. (static call)		Y	Y	Y				Y
	Double (static call)		Y		Y				Y
	createReference/deleteReference		Y		Y		Y		Y

- Array and OrderedArray: Babel's array object allows arrays to be declared specifically as row major or column major ("ordered array") or to be defined implicitly by the strides through memory. Ordered arrays require additional checking as they are passed, and may require translation from the ordering in which they are presented into the ordering requested by the callee. In our tests, no translation was required.
- Static calls: Tests the difference between invoking functions in their static forms, i.e., `Class::function()`, rather than via an object pointer, i.e., `object_ptr->function()`.
- createReference/deleteReference: Tests the cost of Babel's reference counting mechanisms.

In the case of native C++ and Babel, both concrete and virtual function calls were tested.

Because the total duration of an empty function call (where the function merely returns, doing no work) is so short, our approach was to measure the cost of repeatedly calling the function within a loop relative to the cost of a matching empty loop. We used a range of iteration counts (1,000 to 8,192,000 by factors of 2) to ensure that our measurements scaled appropriately, and at each iteration count we took the minimum time from ten consecutive trials. While efforts were made to minimize the interference with these timing runs by running in single-user mode with a minimum of operating system services active, some interference is inevitable. The per-call overheads we report represent an average over the 14 different iteration counts and we estimate that they are generally reliable to $\pm 10\%$. Because the overhead of any specific application function call will depend on the function's arguments, and our primary interest in these timings is the costs of the CCA environment *relative to* the native language costs, we do not consider the observed variability to be a serious issue.

Measurements were carried out on a 500 MHz Pentium III (Coppermine) Dell Latitude CSx laptop running Debian's "unstable" GNU/Linux distribution and 2.4.18 Linux kernel. Version 2.95.4-15 of the GNU compiler toolchain was used, along with Ccaffeine version 0.3, Babel version 0.7.1 (a prerelease of 0.7.2), and omniORB version 3.0.4. The `-O2` flag was used to optimize all compiled code. The `gettimeofday` system function was used for the timing. This function returns wall clock time rather than CPU utilization, as `getrusage` does, but tests showed that despite

Table 2: Actual timings for F77 function calls and relative costs for other environments. Results represent the average across the group or the range (minimum–maximum) where there is significant variation ($>\sim 10\%$) within the group.

Function Group	F77 Time (ns)	C Rel. F77	C++ Rel. F77	Babel C to C Rel. F77	Ccaffeine Rel. F77	OmniORB Rel. F77
A	18	1.0	1.2	2.6	2.4	91.1
B	10–16	1.0–2.2	2.4–3.8	3.2–3.9	3.5	130.8
C	18	1.1	1.1–3.7	2.1–14.4	2.2–4.3	90.8
Overall Average	17	1.1	1.8	3.8	2.8	97.6

Table 3: Timings for Babel interlanguage function calls, relative to the Babel C to C and native F77 timings, according to the function groupings in Table 1. Results represent the average across the group or the range (minimum–maximum) where there is significant variation ($>\sim 10\%$) within the group.

Calling	Called	Timing Rel. Babel C to C					Timing Rel. F77			
Lang.	Lang.	A	B	C	D	Avg.	A	B	C	Avg.
C	C	1.0	1.0	1.0	1.0	1.0	2.6	3.2–3.9	2.1–14.4	3.8
C++	C	1.3	1.3–1.5	1.5–45.3	1.3–7.2	4.9	3.5	4.0–5.8	4.0–21.6	6.3
F77	C	1.0	1.1	0.94–33.6	0.91–1.1	4.3	2.7	3.4–4.1	2.5–41.2	7.3
C	C++	1.5	1.5	1.7–41.1	1.5–13.7	6.9	3.9	4.7–5.8	4.6–57.5	12.2
C++	C++	1.9	1.8	2.2–84.3	1.9–20.4	10.8	4.9	5.4–7.6	6.2–56.5	14.5
F77	C++	1.6	1.5–1.8	1.7–70.9	1.8–14.5	10.0	4.1	5.9	5.2–91.8	15.3
C	F77	1.6	1.7	1.0–90.1	1.8	8.9	4.1	6.1	3.9–61.1	10.3
C++	F77	1.8	2.2	1.5–132	2.2–8.1	12.8	4.9	7.1–8.3	5.8–65.6	13.3
F77	F77	1.7	1.6–2.1	1.0–121	1.8–2.2	12.2	4.4	6.5	4.4–103	14.2

reporting times down to the microsecond, the Linux implementation of `getrusage` had a resolution of only 10 ms, whereas testing indicated `gettimeofday` provides 2 μs resolution.

5 Results and Discussion

5.1 Native Language Results

Table 2 displays the per-call function costs for calls in the C, C++, and F77 native language environments. Within a language, variation among the single-argument functions is generally small. In C, we represented complex values by structures, and when passed by value, these cost roughly 1.2$\times$ most of the other C or F77 function calls. Also, the C function with no arguments that returned a double cost 2.2$\times$ the Group A result. In C++, the function calls with no arguments (with and without a return value), and those with boolean and string arguments, were relatively expensive, from 2.4$\times$–3.8$\times$ the corresponding F77 timings. The C++ results shown are for concrete function calls; virtual calls are uniformly twice as expensive, and because of its implementation, are represented by the Ccaffeine results.

5.2 Babel

Table 3 shows the costs of various interlanguage calls within the Babel environment. Because there are a number of function calls possible in the Babel environment that are not possible in the native environment, we present results relative to both native F77 and the Babel C to C timings. All results are for concrete function calls. Virtual function calls in Babel have essentially the same cost, 1.02$\times$ the concrete call, averaged over all functions and all language combinations. Given the significant variations seen in some of the timings, the overall averages presented can be considered only as a very rough guide for comparisons — it is important to consider both the languages involved and the function arguments when comparing Babel results.

Table 4: Costs for individual Group C functions (see Table 1) for various language combinations in the Babel environment. Costs are relative to the Babel C to C results. For each column, the top label is the *calling* language and the bottom label is the *called* language.

	C	C++	F77	C	C++	F77	C	C++	F77
	C	C	C	C++	C++	C++	F77	F77	F77
Argument	Time (ns)	Time Rel. Babel C to C							
Array	44.3	3.8	1.0	8.7	11.4	8.7	1.6	4.6	1.8
Bool	43.7	2.5	1.9	2.2	3.3	2.7	2.4	3.5	3.0
Complex (by value)	49.8	1.5	1.2	1.7	2.3	1.9	1.6	4.0	1.5
Double Complex (by reference)	45.0	3.0	1.1	2.3	5.8	3.1	1.6	4.0	1.5
Double Complex (by value)	57.3	2.2	0.94	1.9	3.5	1.7	1.6	3.3	1.5
OrderedArray	255	1.5	1.0	1.7	2.2	1.7	1.0	1.5	1.0
String (by reference)	43.8	45.3	33.6	41.1	84.3	70.9	90.1	132	121
String (by value)	39.0	1.9	19.5	27.3	26.8	43.5	29.0	31.5	49.2

We can see that calls involving C tend to be the least expensive. This is not surprising, given Babel's interal object representation is implemented in C. Calls involving C++ tend to be the most expensive, because Babel tries to provide arguments as close as possible to the native language form, and C++ requires the most adaptation. We see that Group A and B functions are generally fairly consistent in cost across the various language combinations, at most 2.2× the C to C cost. Groups C and D show rather large variations in timing and are largely responsibly for driving up the overall average figures. In analyzing the relative costs of functions in Group C especially, it is important to consider them individually — the overall averages, or even the Group C ranges given, can be no more than a very rough guide.

Table 4 shows details of the costs of Group C functions. The most striking feature of these results is tremendous variation in the cost of passing strings, either by value or by reference. This is because in most cases, Babel must allocate new space (via `malloc`) and copy the string as part of adapting it from one language to the other. Most other functions show trends much more in line with the results for Groups A and B, though there are certain cases where the required adaptations are somewhat more expensive.

It is also worth noting that, thus far, development of Babel has focused almost entirely on correctness of the implementation and on expanding the base of languages supported — little effort has gone into optimization. Therefore, we can anticipate improvements in some of these results.

5.3 Native Languages, CCA, and CORBA

In addition to the native language results previously discussed, Table 2 shows the cost of various function calls in the Ccaffeine CCA framework, the Babel environment calling from C to C, and the omniORB CORBA framework relative to the native F77 timings.

As previously noted, an inter-component function call in a direct connect CCA framework, such as Ccaffeine, is equivalent to a C++ virtual function call. The cost is roughly 2.8× the cost of a native F77 function call.

Taking full advantage of the CCA environment — using Babel integrated into a CCA Framework — calls would incur the cost of a virtual function call in the Babel environment, which is practically identical to the cost of a concrete function call.

To gauge the cost of the CCA environment relative to a typical CORBA environment, we also present timings for same-process calls using omniORB. Timings were quite consistent within each group, and the overall average is that the CORBA calls take 97.6× a native F77 call. This is 34.9× the cost of calls in the Ccaffeine CCA framework, and roughly 25.7× the cost of the stand-alone Babel or full CCA (Babel integrated into a framework) environments.

As discussed earlier, these overheads will noticeably effect only the small fraction of functions which are called many times and contain very little work. These results can be used by software architects and component developers to help gauge which functions, if any, are likely to require special consideration in the design of their interfaces. A variety of options are available, depending on the specific situation. For example, if performance is more important than language interoperability, it may be desirable to eliminate the Babel layer for selected component interfaces. If there is flexibility in the overall architecture, it might be modified to make the sensitive function calls intra-component rather than inter-component, thus eliminating the framework overhead. It is worth noting that CORBA does not

provide this kind of flexibility to developers.

6 The Future of the CCA

The CCA is currently at the stage of a highly functional prototype environment. The specification is nearly complete and is more than adequate to enable serious scientific simulations to be developed. A number of prototype frameworks exist, each focusing on different environments (i.e., parallel, distributed, etc.). A variety of mini-applications have been demonstrated using these frameworks, abstracted from real scientific simulations [25], and more than 15 groups have already adopted the CCA as the basis for new terascale scientific simulations which are now under development.

Development of the CCA continues and is accelerating, thanks especially to the formation of the Center for Component Technology for Terascale Simulation Software (CCTTSS) [3] with funding from the U. S. Dept. of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative [8]. The CCTTSS, a subset of the CCA Forum that includes participants from six national laboratories and two universities, carries out research in component technology for high-performance computing and will develop the CCA into a full production-quality environment.

One focus area of the Center is the development of a suite of components based on popular numerical and other libraries in order to "seed" the development of a component-rich environment. Related to this is the the development of domain-specific "standard" interfaces to facilitate the creation of reusable and interoperable components. As such activities are best undertaken by experts in the relevant domain, the role of CCTTSS and the CCA Forum is primarily to encourage and promote the formation of communities around such efforts. Efforts are already underway to develop interfaces for basic scientific data objects, such as distributed arrays, structured and unstructured meshes, and adaptive mesh refinement.

Another focus of the CCTTSS, and one with more performance considerations, is the development of general interfaces and tools for parallel data redistribution, especially for the case of coupling parallel models running on differing numbers of processors. While the initial implementation of this capability will be based on components, there is longer-term interest in doing this at the framework level through more expressive interfaces able to capture the desired parallel semantics, leading to what might be termed *parallel remote method invocation*.

7 Conclusions

The CCA provides a means for developers to manage the complexity of large-scale scientific software systems, and to move toward a "plug and play" environment for high-performance computing. The CCA model allows for a *direct connection* between components within the same process, maintaining performance on inter-component calls. It is neutral with respect to parallelism, allowing components to use whatever means they desire to communicate within their parallel *cohort*. The current prototype CCA environment is being used to create serious scientific simulations, and is also being refined toward production quality. Performance concerns will continue to be central to the development of the CCA.

8 Acknowledgments

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom are gratefully acknowledged.

This work has been supported by the U. S. Dept. of Energy's Scientific Discovery through Advanced Computing initiative, through the Center for Component Technology for Terascale Simulation Software, of which LLNL and ORNL are members.

References

[1] CCA Documents. `http://www.cca-forum.org/documents/`.

[2] CCA Forum home page. `http://www.cca-forum.org`.

[3] Center for Component Technology for Terascale Simulation Software (CCTTSS) home page. http://www.cca-forum.org/ccttss/.

[4] Components @ LLNL: Babel. http://www.llnl.gov/CASC/components/babel.html.

[5] Message Passing Interface (MPI) Forum home page. http://www.mpi-forum.org.

[6] Open Visualization Data Explorer. http://www.opendx.org.

[7] PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/.

[8] Scientific Discovery through Advanced Computing (SciDAC) home page. http://www.science.doe.gov/scidac/.

[9] Sierra home page. http://www.cfd.sandia.gov/sierra.html.

[10] VTK home page. http://public.kitware.com/VTK/.

[11] Benjamin A. Allan, Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, in press.

[12] Global Array Toolkit home page. http://www.emsl.pnl.gov:2080/docs/global/.

[13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[14] David L. Brown, Geoffrey S. Chesshire, William D. Henshaw, and Daniel J. Quinlan. OVERTURE: An object-oriented software systems for solving partial differential equations in serial and parallel environments. In Michael Heath, Virginia Torczon, Greg Astfalk, Petter E. Bjørstad, Alan H. Karp, Charles H. Koebel, Vipin Kumar, Robert F. Lucas, Layne T. Watson, and David E. Womble, editors, *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 1997. (Published only on CD-ROM).

[15] Edmond Chow, Andrew J. Cleary, and Robert D. Falgout. Design of the HYPRE preconditioner library. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 106–116. Society for Industrial and Applied Mathematics, 1999.

[16] John de St. Germain, Steve Parker, John McCorquodale, and Chris Johnson. Uintah: A massively parallel problem solving environment. In *9th IEEE international Symposium on High Performance Distributed Computing (HPDC-9, 2000)*, pages 33–42. IEEE Computer Society, 2000.

[17] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, November 1994.

[18] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*, volume 2 – The MPI-2 Extensions. MIT Press, September 1998.

[19] Object Management Group. OMG's CORBA website. http://www.corba.org.

[20] S. Karamesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in POOMA II. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science. Springer, 1998.

[21] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2001. Also available at http://www.llnl.gov/CASC/components/publications.html.

[22] V. Matena, M. Hapner, and B. Stearns. *Applying Enterprive JavaBeans: Component-Based Development for the J2EE Platform*. The Java Series. Addison-Wesley, 2000.

[23] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing'94*, pages 340–349, Los Alamitos, California, USA, 1994. Institute of Electrical and Electronics Engineers and Association for Computing Machinery, IEEE Computer Society Press.

[24] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *J. Supercomputing*, 10(2):169, 1996.

[25] Boyana Norris, Satish Balay, Steve Benson, Lori Freitag, Paul Hovland, Lois McInnes, and Barry Smith. Parallel components for PDEs and optimization: Some issues and experiences. Technical Report ANL/MCS-P932-0202, Argonne National Laboratory, February 2002. Available via `http://www.mcs.anl.gov/cca/papers/p932.pdf`; under review as an invited paper in a special issue of *Parallel Computing* on Advanced Programming Environments for Parallel and Distributed Computing.

[26] Manish Parashar and James C. Browne. Systems engineering for high performance computing software: The HDDA/DAGH infrastructure for implementations of parallel structured adaptive mesh refinement. In S. B. Baden, N. P. Chrisochoides, D. B. Gannon, and M. L. Norman, editors, *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, volume 117 of *The IMA Volumes in Mathematics and its Applications*, pages 1–18. Springer, 2000.

[27] Manish Parasher, James C. Browne, Carter Edwards, and Kenneth Klimkowski. A common data management infrastructure for adaptive algorithms for PDE solutions. In *SC97 Conference Proceedings*. Association for Computer Machinery and IEEE Computer Society, November 1997. `http://www.supercomp.org/sc97/proceedings/`.

[28] John V. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Kataryzna Keahey, M. Srikant, and MaryDell Tholburn. POOMA: A framework for scientific simulations on parallel architectures. In Gregory Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

[29] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[30] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1 – The MPI Core. MIT Press, 2nd edition, September 1998.

[31] Advanced Visualization Systems. `http://www.avs.com`.

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551

UCRL-JC-150544

Component Technology for Laser Plasma Simulation

W. J. Bosl, S. G. Smith, T. Dahlgren, T. Epperly, S. Kohn, and G. Kumfert

September 23, 2002

U.S. Department of Energy
Lawrence Livermore National Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Component Technology for Laser Plasma Simulation

William J. Bosl
bosl@llnl.gov

Steven G. Smith
sgsmith@llnl.gov

Tamara Dahlgren
dahlgren1@llnl.gov

Thomas Epperly
epperley@llnl.gov

Scott Kohn
kohn1@llnl.gov

Gary Kumfert
kumfert@llnl.gov

Lawrence Livermore National Lab
P.O. Box 808, L-561
Livermore, CA 94551

ABSTRACT

This paper will discuss the application of high performance component software technology developed for a complex physics simulation development effort. The primary tool used to build software components is called Babel and is used to create language-independent libraries for high performance computers. Components were constructed from legacy code and wrapped with a thin Python layer to enable run-time scripting. Low-level components in Fortran, C++, and Python were composed directly as Babel components and invoked interactively from a parallel Python script.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability – *distributed objects, interface definition languages.*

General Terms

Algorithms, Performance, Languages

Keywords

Components, scientific computing, numerical methods, physics.

1. INTRODUCTION

The scientific computing community has invested a significant amount of resources towards the development of high-performance scientific simulation software, including numerical libraries, visualization, steering, software frameworks, and physics packages. Unfortunately, because this software was not designed for interoperability and re-use, it is often difficult to share these sophisticated software packages among applications due to differences in implementation language, programming style, or calling interfaces. It is highly desirable to be able to reuse large and complicated software packages without having to devote large amounts of time to re-engineer them [1]. Moreover, many of the simulations that are required today involve multiple physical and chemical processes, so-called multiphysics simulations. Building these codes from pre-tested software components is much more reliable and efficient than trying to build a complete simulator from scratch [2].

One example of a complicated multiphysics simulation problem is the interaction of lasers with plasmas. Simulation of laser plasma interaction is an important design tool, complementing theoretical analysis and experimentation for developing complicated laser tools for studying inertial confinement fusion. The software required for simulating these complex physical processes reflects the physical system: it is complex. To carry out numerical experiments and analyze the resulting computational data, the software must be flexible enough to allow scientists to quickly and easily compare competing physics models and alternative design strategies. Constructing complex simulation codes from available software components is an efficient strategy for building a new laser plasma simulation code.

In this paper, will present our experiences wrapping a large scientific simulation code using the Babel language interoperability tool [8] so that the application could be driven from the Python scripting language. Furthermore, we were able to freely mix C++, Fortran, and Python modules in the software. For example, from the scripting layer, we were able to call the application code in C++, which in turn called a numerical routine written in Fortran, which in turn called a bounary condition routine written in Python. This language interoperability enabled us to rapidly prototype new boundary conditions modules in Python without recompiling or linking the whole code. We discovered that compiler incompatibilities introduced some difficulties in code reuse. This problem is ubiquitous and is not limited to the Babel tool. We will discuss the trade-offs using a tool such as Babel as compared to a more traditional wriapping solution such as SWIG.

2. ALPS: Adaptive Laser Plasma Simulator

The ability to predict and control laser-plasma interactions is critical for the design of inertial confinement fusion (ICF) experiments. ICF involves the use of high powered lasers to rapidly ionize and compress hydrogen fuel pellets sufficiently to initiate a fusion reaction. During these experiments, a plasma filled region is created by the ionizing fuel. The laser must continue to

propagate through the plasma region to achieve the desired distribution of energy at the target fuel pellet. Simulation of the laser plasma interactions is used to predict and control laser parameters for ICF experiments.

The Adaptive Laser Plasma Simulator (ALPS) project [3] is being developed using the SAMRAI (Structured Adaptive Mesh Refinement Applications Infrastructure) [4,5] system currently under development in CASC. SAMRAI is a C++ class library that supports the development of application codes utilizing structured adaptive mesh refinement (AMR) algorithms. Parallelism on distributed memory architectures is handled by the framework, freeing the user from most of these details. Data layout and interprocess communication is performed through an interface to the standard Message Passing Interface (MPI) library.

3. Component Software Technology

Component technology is an extension of scripting and object-oriented software development techniques that specifically focuses on the needs of software re-use and interoperability. Component-based software techniques address issues of language independence and component connection behavior that other software techniques do not address. To use a hardware analogy, a component is like a "software integrated circuit" with well-defined pin-outs that may be connected to compatible pins on other "software integrated circuits." Figure 1 is a cartoon illustration of how we used Babel as the backplane to connect software components together to create an application.

3.1 Commercial solutions

Component approaches based on CORBA [9], COM [12], and Java technologies are widely used in industry but will not scale to support large parallel applications in science and engineering. Our research focuses on the unique requirements of scientific computing on high-performance machines, such as fast in-process connections among components, language interoperability for scientific languages, and data distribution support for massively parallel SPMD components.

3.2 Babel

Babel is a language interoperability tool that uses a *Scientific Interface Definition Language* (SIDL) to describe component interfaces. Using SIDL descriptions, Babel automatically generates code to mediate differences between components written in different languages.

Computational scientists developing large simulation codes often face difficulties due to language incompatibilities among various software libraries. Scientific software libraries are written in a variety of programming languages, including Fortran, C, C++, or a scripting language such as Python. Language differences often force software developers to generate mediating glue code by hand. In the worst case, computational scientists may need to re-write a particular library from scratch or not use it at all. We have developed a tool called Babel that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation, description, and distribution of language independent software libraries.

Babel addresses the language interoperability problem using Interface Definition Language (IDL) techniques. An IDL describes the calling interface (but not the implementation) of a particular software library. IDL tools such as Babel use this interface description to generate *glue code* that allows a software library implemented in one supported language to be called from any other supported language. We have designed a Scientific Interface Definition Language (SIDL) that addresses the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to Java.

The Babel parser, which is available either at the command-line or through the Alexandria web interface, reads SIDL interface specifications and generates an intermediate XML representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a repository or a problem solving environment. XML interface descriptions are stored either in a local file repository or on the web using Alexandria. The vision is that a scientist downloading a particular software library from the component repository will receive not only that library but also the required language bindings generated automatically by the Babel tools.

The Babel code generator reads SIDL XML descriptions and automatically generates glue code for the specified software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space and, eventually, across memory spaces for distributed objects. The code generators create four different types of files: stubs, skeletons, Babel internal representation, and implementation prototypes. The Babel internal object representation created by the code generators is similar to that used by COM, CORBA's Portable Object Adaptor, and scientific libraries such as PETSc. The internal object representation is essentially a table of function pointers, one for each method in an object's interface, along with other information such as internal object state data, parent classes and interfaces, and Babel data structures. Stub and skeleton code translates between the calling conventions of a particular language and the internal Babel representation. The code generators also create implementation files that contain function prototypes to be filled in by the library developers. To simplify the task of library writers, we have added automatic Makefile generation as well as a *code splicing* capability that preserves old edits during the regeneration of implementation files after modifications to the SIDL source. Finally, the run-time library provides general services such as reference counting and dynamic type

identification. In the future, we expect to support dynamic loading of objects, reflection, and a dynamic invocation interface.

4. PyALPS

Currently, our laser plasma simulations are carried out using a uniform rectangular grid. This prohibits the use of high resolution in the regions of greatest interest by requiring a uniform grid over the entire domain. However, the code currently used for laser-plasma simulation is highly developed as a scientific and engineering design tool. In particular, an in-house scripting language called Yorick [11] is used for interactive steering and control of laser calculations. Yorick is an interpreted programming language, designed for postprocessing or steering large scientific simulation codes. Smaller scientific simulations or calculations can be written as standalone yorick programs. The language features a compact syntax for many common array operations, so it processes large arrays of numbers very efficiently.

4.1 Scripting

For use as a scientific and engineering design tool, ALPS requires the run-time flexibility of a scripting language, such as the Yorick capability that current laser physicists are accustomed to having. We adopted Python as a scripting language because it has a large and growing scientific user base and has a parallel implementation.

Since detailed simulations of laser plasma interactions can consume many hours of supercomputer time, it is often desirable to do calculations with either limited spatial resolution or a small number of time steps, then look at the results and determine whether some adjustment of the parameters is needed before continuing on with a lengthy calculation. Similarly, short period simulations may be used to examine the effects of parameter variations. Scripting enables laser scientists to perform simulations in a controlled fashion to maximize the amount of information that can be obtained in a limited time [8]. It also allows a great deal of flexibility by allowing different or new physics modules to be invoked quickly and easily. Scripted codes can be run interactively or in batch mode, giving the user considerable flexibility over a simulation.

We have used Babel to develop a scripted version of ALPS that uses Python as the scripting language. Wrapping parts of the ALPS code using Babel enables the creation of plug-n-play modules in a variety of supported languages. From the highest level at which users interact with pyAlps, the ALPS application appears to be a Python package, consisting of pure Python modules. that enables application users to compare ALPS results against those produced by an existing computational tool. The scripted interface will also allow ALPS users to interact with a running simulation to visualize data on-the-fly. This collaboration is the first to demonstrate Babel's applicability in a large-scale scientific application.

One of the primary goals of creating a scripted version of ALPS was to enable users to run ALPS interactively. Babel was used to create thin Python wrappers for important capabilities in the ALPS code. Specifically, we wrote interface files with Babel's Scientific Interface Definition Language (SIDL), which is similar to the IDL interface used to write CORBA interfaces. The SIDL file is a language-independent, object oriented description of the attributes (member variables) and methods associated with interfaces and classes. Babel uses the information in the SIDL file to create language bindings for any of the supported languages.

An example of a SIDL file is shown here. It contains class definitions for the basic Alps class and for beam modules, which compute the energy intensity contained in a laser beam. The SIDL file is used by the babel software to generate client-side and server-side code, each in a specified language. For the Alps class, the client is written in Python and all relevant files are presented to the user as the pyAlps package. Once imported as a Python package, an Alps class is created and methods can be invoked. After initialization from an input file or restart data file, the user may invoke several different run options in order to control time stepping precisely. Visualization files can be written at any point after the simulation has run to the currently-specified time and viewed using visualization software. Parameters can be adjusted using Python-wrapped database manipulation methods for the input variables.

The following code is an example of a SIDL file for the pyALPS package. Babel uses the information in this file to create glue code in any of the supported languages to wrap each of the specified objects.

```
version pyAlps 0.1;
package pyAlps {
  class Alps {
     void initialize(in pySAMRAI.InputDatabase database);
     void initializeFromRestart(in string dir, in int num, in
                          pySAMRAI.InputDatabase database);
     double run(in double time);
     double runToFinish();
     double runTo(in double time);
     double step(in int num_iter);
     double stepTo(in int iteration);
     void writeRestart(in string fname, in int seq_num_ext);
     void writeVis(in string fname, in int seq_num);
     void finalize();
  }
  abstract class Beam {
     abstract void setBeam0(inout array<dcomplex,2> amp);
     final void setDopplerShift(in double a_doppler_shift);
     final double getDopplerShift();
     final void setCenter(in array<double,1> a_center);
     final void getCenter(out array<double,1> a_center);
     final void setMaxIntensity(in double a_intensity);
     final void getMaxIntensity(out double a_intensity);
  }
  dass Cos2_Beam extends Beam {
     void setBeam0(inout array<dcomplex,2> amp); }
  class SphericalCos2_Beam extends Beam {
     void setBeam0(inout array<dcomplex,2> amp); }
```

```
class Gaussian_Beam extends Beam {
   void setBeam0(inout array<dcomplex,2> amp); }
class SuperGaussian_Beam extends Beam {
   void setBeam0(inout array<dcomplex,2> amp); }
}
```

In particular, note that the beam class is declared to be an abstract class. This means that at least of the member functions of the beam class is abstract and is not defined within the beam class. Subclasses of the general beam class must define a setBeam0 method. The abstract beam class also declares a number of member functions that will be explicitly defined in the implementation of the beam class. These member functions are common to all subclasses of the beam module, although they may be substituted with new functions in subclasses. Babel can create Beam modules in any of the supported languages, currently including F77, Python, C, and C++ from the SIDL file.

Our initial task was to decompose the ALPS code into components that were appropriate for run-time scripting. The primary tasks performed in the monolithic code were to read and process input data, initialize data structures, loop through a specified time loop, and output data at regular intervals in the time loop. These code segments formed the basic components that were to be controlled from the script.

The ALPS code simulates the interaction of a set of laser beams with a plasma in space and time. The computational grid is a sophisticated adaptive, multilevel grid that is required for high resolution. Often, run-time parameters for the complex simulation runs are not know precisely. Scientists needed a simulation tool that could be run a certain number of time steps, stopped and queried using visualization tools to inspect intermediate field variables, then modified by changing certain key parameters and run forward in time for a fixed interval again. This gave the scientists a steering capability through Python scripting.

A parallel version of Python, pyMPI, developed at LLNL and available publically through SourceForge [12] was adopted for Python scripting. ALPS is built using the SAMRAI framework for adaptive mesh simulations on parallel machines, together with legacy Fortran code obtained from laser physicists. Linear solvers from the PETSc library and HYPRE are available through the SAMRAI framework and invoked for solving linear systems. Babel was able to generate code to glue together all these packages in appropriate components. Of particular note was the decomposition of SAMRAI into components for data I/O and mesh initialization. From the scientist's view, pyALPS looks like a normal python script. An example of a pyALPS script is shown here:

```
import sys
import pySAMRAI.InputDatabase
import pySAMRAI.Alps

# Create the input database
inputdb = pySAMRAI.InputDatabase.InputDatabase()
inputdb.initialize("ALPS")
inputdb.parseInputFile("alps.input")

# Create alps object and initialize the state
alps = pySAMRAI.Alps.Alps()
alps.initialize(inputdb)

# Change some values
griddingdb = inputdb.getDatabase("GriddingAlgorithm")
print("Old efficiency_tolerance = %f" %
griddingdb.getDouble("efficiency_tolerance"))
print("Old combine_efficiency = %f" %
griddingdb.getDouble("combine_efficiency"))
griddingdb.putDouble("efficiency_tolerance", 0.90)
griddingdb.putDouble("combine_efficiency", 0.90)
print("New efficiency_tolerance = %f" %
griddingdb.getDouble("efficiency_tolerance"))
print("New combine_efficiency = %f" %
griddingdb.getDouble("combine_efficiency"))

# Step 5 time steps …
alps.Step(5)
# … then do something with the data!
# Run to the end specified in the input file
alps.runToFinish()
# Finalize everything
alps.finalize()
```

One of the primary difficulties encountered in this project was related to the need to create dynamic libraries for run-time loading. Incompatible compiler options seemed to cause the most build problems. During the integration process the low level details of simply building the code caused an unexpected number problems. Several of the packages we were integrating had not been compiled as a shared library before. This mandated a reworking of the build systems in order to support the necessary compilation steps. While this was expected, the brittleness of the build process was not. We found that even slight variations in the compiler options used to compile each package could cause link or runtime failures.

The runtime failures in particular are troublesome since a method invocation would fail in a system library for no obvious reason. To overcome this we standardized on a set of compilers and compile flags for all packages. While this is a simple (and obvious) solution, it is not a satisfactory solution if the goal is to have a large set of easy to use components for widespread use. Given the target audience for a scientific component architecture contains developers for whom dynamic linking will be a new experience, these types of problems could pose a barrier for software reuse, especially for software in object or component form. The component software community may need to move towards some kind of compiler meta-data for packages or something else to facilitate mixing of binary libraries, especially with C++.

Creating SIDL files needed to wrap each of the components is is a little tedious, but is relatively straight-forward. We did not find this to be a particularly difficult issue.

4.2 Plug and Play Modularity

In addition interactive control of simulations, the capability of easily swapping in alternative physics modules is a desirable new feature for laser plasma simulations. Scientific investigation using simulation often involves testing and comparing alternative physics modules or new algorithms. Our goal was to enable rapid replacement of classes, subroutines, or groups of related classes and subroutines with alternatives.

To do this, appropriate pieces of code were wrapped using Babel and made into Babel components. These components can be accessed by driver routines written in any of the Babel supported languages. Alternative components can then be written by application scientists in any language that's convenient and wrapped with Babel to make an alternative component that can be seamlessly interchanged with the original component. Because the application scientist is free to implement new components in a language such as Python, new algorithms can be written quickly and tested in the pyALPS code. Important components can be optimized in another programming language later if desired.

One of the novel and powerful capabilities provided by Babel components is the ability to call any of the supported languages from any other. Thus, not only can Python call C or Fortran subroutines as, for example, SWIG extensions to Python, but Fortran can also call Python functions. We used this feature to create a powerful plug and play capability for scientific exploration of new beam modules.

4.2.1 Beam Modules

In the ALPS code, beam calculations are invoked from within the legacy Alps code. The original beam subroutines are written in Fortran and are called from Fortran subroutines, which are originally invoked from the Alps C++ driver code. Using the SIDL file shown above for the Beam class, we made Beams a component of the system and modified the ALPS driver to call Beam components rather than the original embedded Fortran subroutines. Beam modules clients were created in Fortran using Babel to enable us to use the original Fortran beam calculations. Once this was done, we also created Python beam clients to demonstrate this capability. The advantage of Python beam modules is that they can be created quickly and do not need to be compiled to be invoked by the pyAlps simulator. This provides a versatile tool for scientific experimentation.

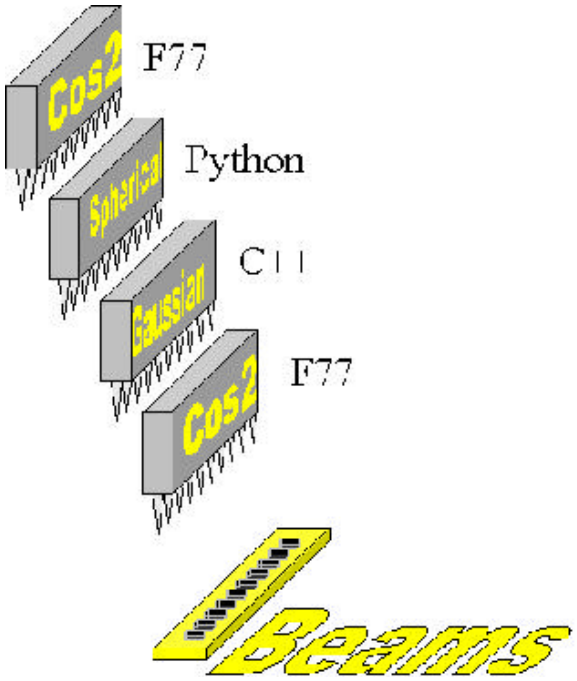

Figure 1. Physics components can be written in any of the languages supported by Babel. Components written in Python, for example, can be invoked without recompiling to rapidly test new algorithms.

4.2.2 Lessons Learned

Creating new modules was not as difficult as building the components created from legacy code and linking them together. This is due largely to the fact that new beam modules are designed and written specifically for the component system. Python modules are particularly easy to write and invoke from the Python script. Perhaps the only difficulty in adopting this approach was learning to use Babel arrays within Fortran in order to pass them to the component layer on the client side. Arrays must be passed back and forth to client and server in a language independent fashion and this is accomplished by requiring the creation of Babel arrays in all user code.

5. Discussion

Several different approaches are available today to build language independent components that can be re-used in multiple applications, used to assemble complex multi-physics simulators from pre-built software, and run simulation codes from a scripting language such as Python. Babel is a tool that offers certain unique features if those features are required, including a powerful array syntax, support for complex numbers, and parallel computing. The price for this capability is a need for careful attention to compiler options for all codes that must interoperate and the need to learn Babel data structures and the Babel scientific interface definition language. If Babel's unique features are required, then this is a price that has to be paid, for there are few other options at this time that provide all these features.

6. ACKNOWLEDGMENTS

Funding for this project is provided by the LLNL Laboratory Directed Research and Development program and the DOE Office of Science.

7. REFERENCES

[1] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois Mcinnes, Steve Parker, and Brent Smolinski, "Toward a Common Component Architecture for High Performance Scientific Computing," *High Performance Distributed Computing Conference*, 1999.

[2] A. Cleary, S. Kohn, S. Smith, B. Smolinski, "Language Interoperability Mechanisms for High-Performance Scientific Applications," *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, Yorktown Heights, NY, October 21-23, 1998.

[3] M. Dorr, X. Garaizar, and J. Higginger, "Simulation of Laser Plasma Filamentation Using Adaptive Mesh Refinement", Journal of Computational Physics, **177**, pp 233-263, 2002. See http://www.llnl.gov/CASC/alps.

[4] R. Hornung and S. Kohn, "The Use of Object-Oriented Design Patterns in the SAMRAI Structured AMR Framework," *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998. See http://www.llnl.gov/CASC/SAMRAI.

[5] R. Hornung and S. Kohn, "Managing Application Complexity in the SAMRAI Object-Oriented Framework," *Concurrency and Computation: Practice and Experience* (special issue on Software Architecture for Scientific Applications), 2001.

[6] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. "Divorcing Language Dependencies from a Scientific Software Library," *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[7] J. Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, **IEEE Computer**, March 1998.

[8] *CORBA Components*, Object Management Group, OMG TC Document orbos/99-02-95, March 1999. See http://www.omg.org

[9] B. Smolinski, S. Kohn, N. Elliott, and N. Dykman, "Language Interoperability for High-Performance Parallel Scientific Components," *International Symposium on Object-Oriented Parallel Environments (ISOPE)*, December 1999.

[10] See http://sourceforge.net/projects/pympi.

[11] *The Yorick Home Page*, 2001. See ftp://ftp-icf.llnl.gov/pub/Yorick/yorick-ad.html.

[12] http://www.microsoft.com/com/default.asp.